I. Project Overview

Introduction

This project involves creating a device that reads status information from a vending machine controller. When there is an inventory or warning status that the vending machine owner should be aware of, the device will blink an LED/button. Consumers who press the button will be given a phone number to call and an encrypted number to enter. Upon receiving a valid encrypted number, the system will send back a code that the user can enter into the device to receive a free item from the vending machine.

Background

The idea originated from Travis Smith, owner of Appolis, Inc. The idea was to introduce a cheaper method of vending machine communication. The current methods, he learned from Pepsi distributors, were either expensive or time-consuming. The distributors he spoke with desired a cheaper solution to their problem for communicating with vending machines. Rather than using other methods of technology, his method will use the consumer as the means of getting vending information to the distributors on the status of their machines.

The purpose of this project will be to change how vending machines communicate with suppliers when they need to be serviced or re-filled. This project would propose a more cost-effective solution than wireless communication. The customer at the vending machine will be given a code from a module and a phone number to call and enter the code. In return, the customer will receive code to enter back into the module to receive a free item from the vending machine. This project will encompass encrypting the codes so that customers cannot guess the code for getting free items from the machine. The module will have a keypad and will need to receive the data provided by the machine.

Our system will first communicate with the DEX (Data EXchange) interface of the vending machine. This provides audit data that will allow us to determine if the machine requires servicing. The customer at the vending machine will be given a code from our device and a phone number to call and enter the code. In return, the customer will receive a code to enter back into a keypad on the module to receive a free item from the vending machine.

II. Definitions

The following definitions are used throughout the report. These definitions are a good reference if the reader is unsure of how a word or command is used.

Acronym	Expanded	Meaning
DEX	Data Exchange	Vending communication protocol
EVA	European Vending Association	Created standards for DEX protocol
CRC	Cyclic Redundancy Check	Check to ensure accuracy of data
DEX COMMANDS:		
ENQ	Inquiry	Decimal 5, Hex 5
ETB	End of Transmission Block	Decimal 23, Hex 17
CR	Carriage Return	Decimal 13
EOT	End of Transmission	Decimal 4, Hex 4
SOH	Start of Header	Decimal 1
STX	Start of Text	Decimal 2
ETX	End of Text	Decimal 3
NL	New Line Feed	Decimal 10

III. Requirements

Our project requirements were written from a meeting with our client, Travis Smith. We learned through the process of designing the project that some of the features he wanted "if possible" could not be implemented because of limitations with the DEX interface. One limitation is that we cannot provide full inventory levels.

Objectives

- Notify customer with indicator, code, and number to call
- Information to be transmitted from the machine to the back-end system
    - Worst-case: machine needs servicing based on inventory level of at least 1 selection
    - Next-best: which selections are low
    - Best-case: full inventory level triggering
    - Code should give information to identify the machine
    - Nice to have: Capability for transmitting at certain times
- Automated phone call system verifies code from machine and returns a code for a free item from the machine to be entered into keypad (for this phase, this function will likely be simulated via web page provided by Appolis, Inc.)
- Make sure codes cannot be re-used, and that after free item is issued, sensor will not trigger another indication until the machine has been reset (re-filled).

Client Requirements

- Provide Appolis, Inc. with encryption algorithm for decrypting the code from the device in C# (C-sharp).
- Use Visual Studio Express 2005

Functions

- Will need to communicate and transmit to machine
    - DEX is the communication standard for data being transmitted from the machine
- Dispensing the free item

Design Constraints

- No more than 16 numbers in code (4 sets of 4 digits)
- Use a pass-through connector so the DEX port isn't unavailable (Nice to have)
- Size
    - Want all hardware to fit inside machine except LCD, LED, button, and keypad, connected by a ribbon cable or equivalent
- Object oriented programming, using C# if at all possible

IV. Design Process

The project can be broken down into three main components.

A. Vending Machine Communication
B. Encrypt the inventory status received
C. Interface with the user

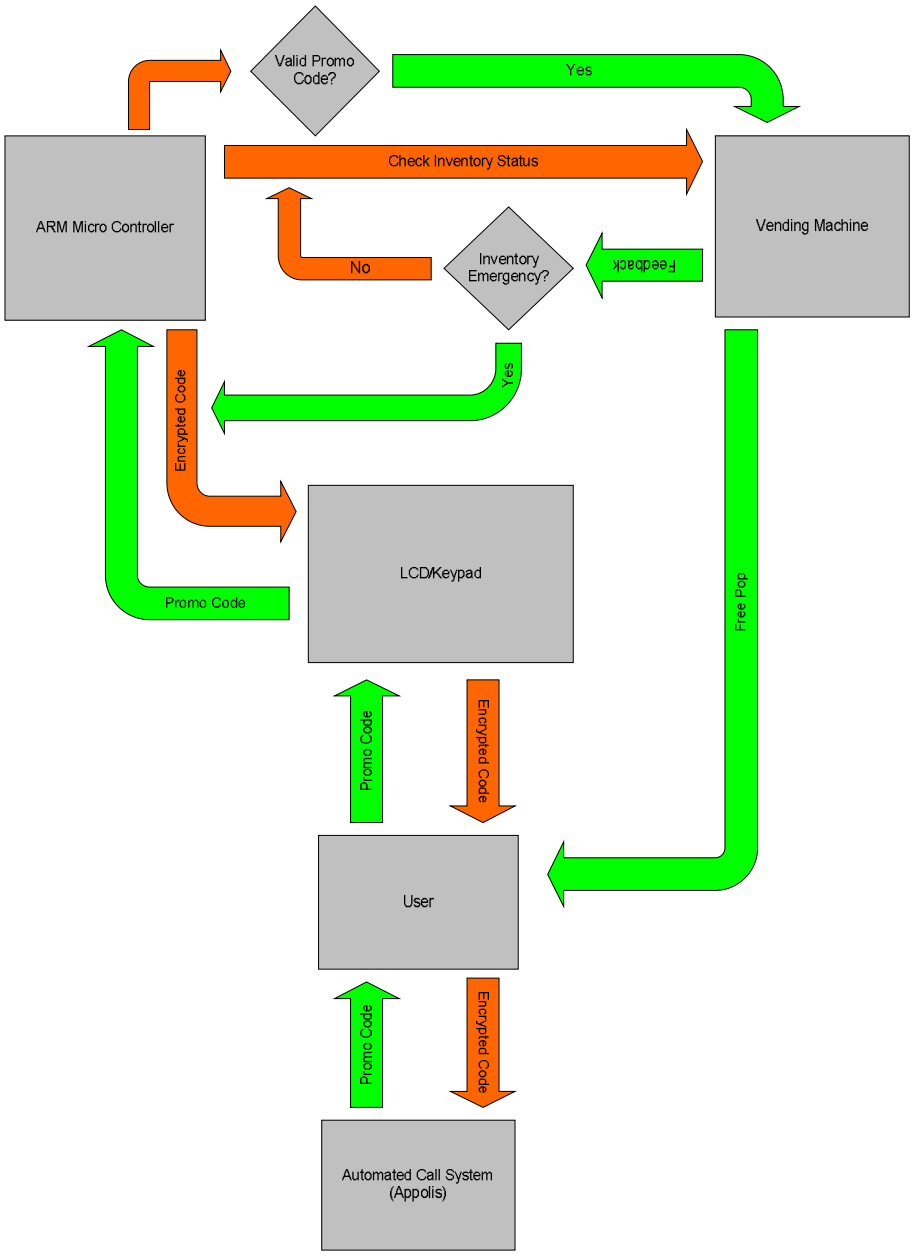


Figure 1 - Block Diagram for the main components of the project

Consumer Interface Flow Chart

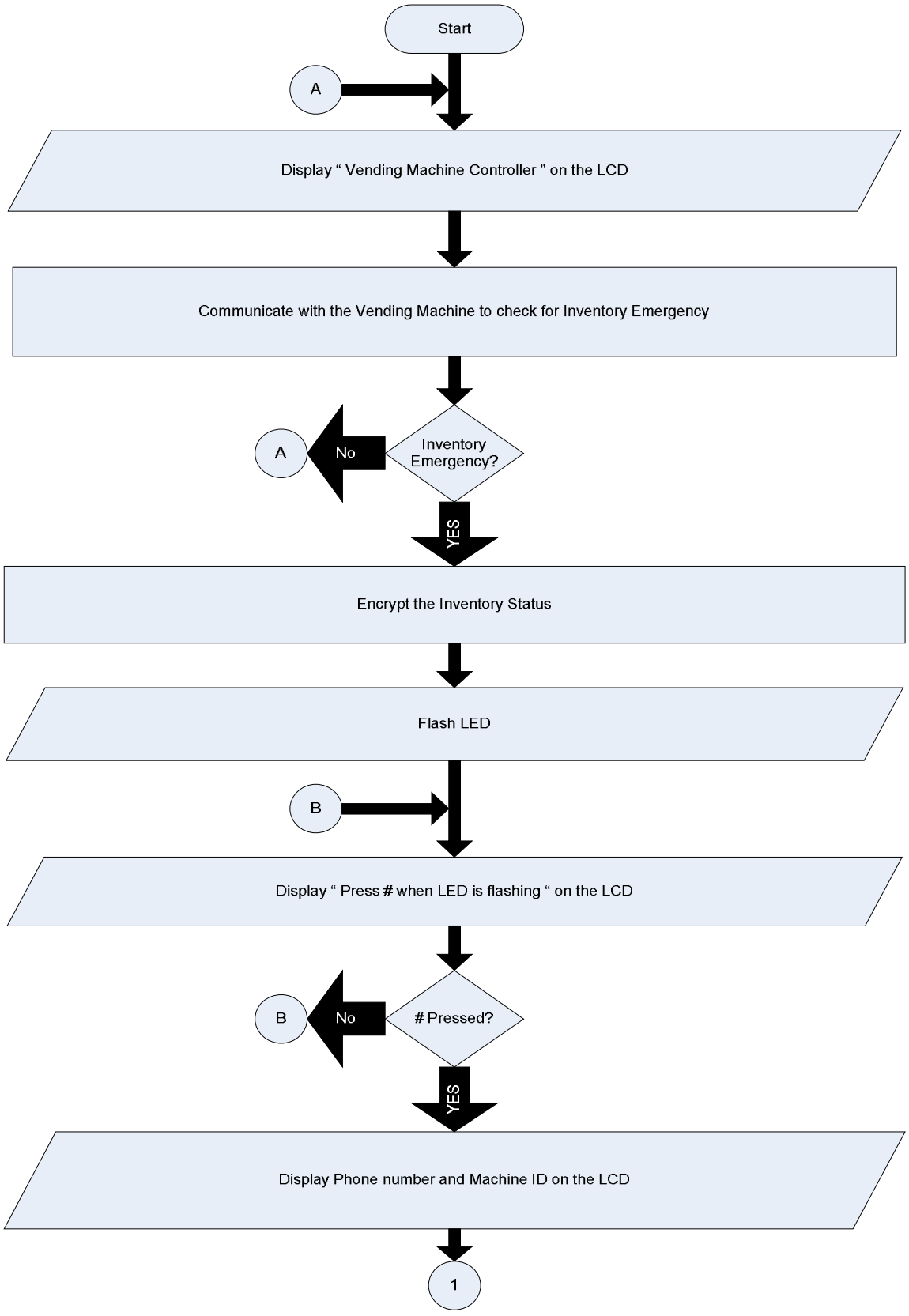


Figure 2 - Consumer interface Flow Chart

Consumer Interface Flow Chart (cont'd)

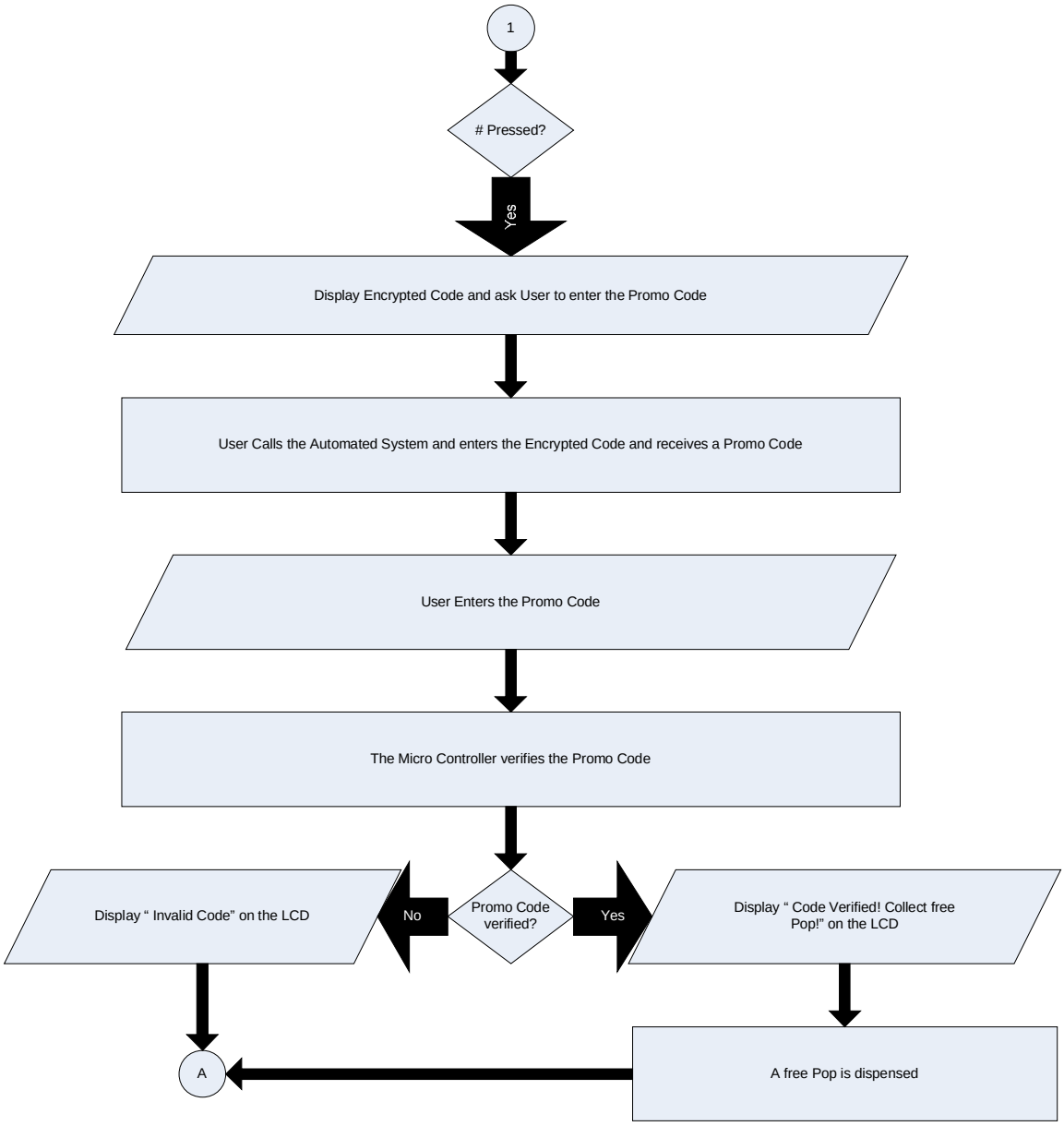


Figure 3 - end of consumer interface flow chart.

A. Vending Machine Communication

For our project, we needed to obtain the audit data from vending machines using the DEX (data exchange) protocol. Serial (RS-232) communication was required to facilitate this communication. Using hardware handshaking, which included three handshake phases, we are able to obtain the required data from the vending machine. Using C# programming with the development board's ARM microcontroller, we successfully stepped through the audit process.

To understand the communication process and to learn the C# programming language, we first used a computer to communicate with the vending machine. Using a console application, we stepped through each part of the process until the end result was achieved. Using this approach we were able to check what is received after each step. For the final product, we will not be able to see each step so this was a great way to test and progress through the communication.

We learned that timing with communication is very important. Thanks to having a reference from the European Vending Association (the EVA DTS, or data transfer standard) we were able to see what timings would work for our use. Along with that standard, C++ code was provided as well as a working console application. Using this application we were able to read the data from our vending machine, so we could understand how different parameters worked. We were not able to copy any of the sample code word for word; we had to understand what each portion of C++ code was doing, then find a way to make that work with C# through the computer. Once done, we needed to transfer that communication code to the ARM micro processor, which also required some changes.

Some limitations we faced with the DEX communication was that only the sold out date and time are provided. This would be much easier if a binary bit would indicate "product sold out". For our first version of this code, we are simply checking the date each product was sold out, and comparing it with today's date, which also appears in the DEX data. For our final product, we will need to put our module in "sleep" mode for a day after the machine has been serviced, since we are limited to days.

The data contained in the audit information has the capability to transmit many types of information. Our machine does not include all possible features, but following will be the items which are obtained and information of how more of this information can be used in the future. For example, if the data can be reset, DEX could be used to provide vending machine distributors with inventory information.

Information obtained from DEX communication:

- Product price, sales
- Product Identification
- Product sold out date and time
- Cash and coin sales amounts
- Machine events
- Machine ID number

Communication with the DEX protocol required making a cable which has a ¼" audio jack on one end, and a serial connector on the other end. The diagram for the cable is shown below in figure 4. We learned how to use our development board (iPac 9302) and ARM microprocessor to communicate with the DEX protocol at a baud rate of 9600 bits per second.

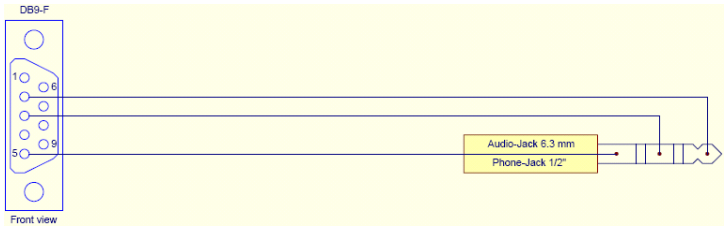


Figure 4 - DEX cable diagram

The DEX commands are shown in figure 5 below. Using the handshaking process these commands were sent and received to get to the data transfer, or third handshake. "ENQ" stands for inquiry and this is how the audit process begins. The flow chart to follow will give a better picture of the process. The DEX commands were saved into single byte arrays in our program. Byte arrays were required to send the data through the serial port. The hexadecimal equivalents are listed next to each command, so we defined each command as the hexadecimal value. Following is an example from our C# code of how we needed to define these commands: *byte [] ENQ = {0x05};*.

Specific control characters and combinations that are permitted are:

01h	SOH	0Ah	LF	10h 30h	DLE 0 (ACK 0)
02h	STX	0Dh	CR	10h 31h	DLE 1 (ACK 1)
03h	ETX	10h	DLE	10h 3Bh	DLE ; (WACK)
04h	EOT	15h	NAK		
05h	ENQ	16h	SYN		
		17h	ETB		

Figure 5 – DEX commands - extracted from European Vending Association Data Transfer Standard 6.1 (EVA-DTS 6.1)

Communication Process

Throughout the flow chart the commands "send" and "receive" refer to data sent and received through the serial port using RS-232 communication. The C# commands for those functions on the development board are as follows:

- *portDex.Write(ENQ, 0, DLE.Length);*
- *portDex.Read(ccbuf, 0, ccbuf.Length, 1000);*

For the "write" command, we simply passed our defined variables, the starting bit, and the array length. Most of our byte arrays were one character long. For the "read" command, we read all the incoming information into a buffer. We used a 150 character byte array for this purpose, "ccbuf". The "1000" is the timeout time, which is not required for serial port communication on the personal computer, but is required with the embedded device. The timeout time will be called if no data is received within the first second, as the data is shown in milliseconds. We used the "sleep" command for delays between writing

and reading to the serial port (for example "*Thread.Sleep(500);"*). The number passed into the sleep command is also in milliseconds. More details are provided in the "Software" section of the report.

The stages for the serial DEX communication are as follows:

- First (master) handshake
- Second (slave) handshake
- Third (data transfer) handshake

The main program is the main function which steps through each part of the process. To enter into the first handshake, a "DLE" must be received from the machine after sending "ENQ" to the machine. If an "ENQ" is received back, this means the machine is asking for the data to be transmitted again. See the "Software" section for detailed information and flow chart.

B. ENCRYPTION ALGORITHM

- Information from the vending machine is acquired through DEX Communication.
- The inventory status is analyzed and the message is generated.
- Message is sent in two steps

    - Machine ID (10)
    - Encrypted code for the Inventory status (8)

- '1' is used to notify an inventory emergency. '0' denotes a normal state.
- If a product is sold-out, it is assigned a '1' and a '0' otherwise.
- Due to memory constraints and limitations of encryption algorithm, we have decided to exclude the Machine ID from the encrypted code.
- The message generated after analyzing the information from the vending machine, is encrypted using Key1.
- The customer is displayed a screen with the Phone number and the Machine ID.
- Then the encrypted code is displayed.
- The Machine ID does not need decryption.
- The encrypted code is decrypted using Key1, giving you the original message.
- The original message is then encrypted using Key2 to generate the Promotional Code for a free vend.
- When the Promo Code is punched in by the customer, it is decrypted using Key2.
- The message so generated should be the same as the original message. This is how the Promo Code is validated and a free pop is dispensed.

Sample info from Vending Machine

If we get the following data from the Vending machine, the code is generated as shown below.

MachineID 7986623432444

The data blocks from the Vending Machine have an Audit Read Date which tells us about the date when the info was sent. The data block also provides the sold out date of each product.
The sold out date for each product is compared with the Audit Read Date and if they are the same, the product is deemed sold out and the cell in the code representing its status is set high.
There is a similar counter to indicate if the cash box or one of the coin tubes is full.

If products 1, 5 and 7 are found to be sold-out, we assign '1' to the respective columns of the original code.
The rest of the products are assigned a '0'.

If the counter for, either the cash box or one or more coin tubes says full/empty, we assign a '1' to the column for coin/cash.
Let's assume one of the coin tubes is found to be full/empty.

Triggering the Encryption Process

Every time we receive the data blocks from the Vending machine, we check if any of the products are sold out. If none of the products are sold out, the micro controller waits for a user specifiable time period and check again. If any of the products are found to be sold out, the encryption process is triggered.

The micro controller analyses the data blocks and generates a code that indicates which products are sold out.
The original code then goes through the following steps.

Step1 Generating the original code (Vending Machine end)

So, we get the following code. This is the code to be sent so that Appolis can get the inventory status update.

	P1	P2	P3	P4	P5	P6	P7	Coin/Cash
Original Code	1	0	0	0	1	0	1	1

Next we assign the encryption/decryption keys.

Key1	1	1	0	0	0	1	1	0

Key2	1	0	1	1	1	1	0	0

Step2 Generating the encrypted code (Vending Machine end)

The Original code is encrypted using Key1. We perform an XOR function on the two arrays to get the encrypted code.

Original Code	1	0	0	0	1	0	1	1

Key1	1	1	0	0	0	1	1	0

Encrypted Code	1	0	1	1	0	0	1	0

This encrypted code is displayed to the customer along with the MachineID.

Step3 Decrypting the Code to get the Original Code (Appolis end)

Upon receiving the encrypted code, it is XOR–ed with Key1 to get back the Original Code.

Encrypted	1	0	1	1	0	0	1	0
Key1	1	1	0	0	0	1	1	0
Original Code	1	0	0	0	1	0	1	1

The relevant information is extracted from the Original Code and stored for further use.

Step4 Generating the Promotional Code to send back to the customer (Appolis end)

Now the Original Code is encrypted using Key2 to generate the Promotional Code.

Original Code	1	0	0	0	1	0	1	1
Key2	1	0	1	1	1	1	0	0
Promotional Code	1	1	0	0	1	0	0	0

Step5 Validation of the Promotional Code (Vending Machine end)

The Promotional Code received is XOR-ed with Key2 to get back the Original Code, and hence validate it.

Promotional Code	1	1	0	0	1	0	0	0
Key2	1	0	1	1	1	1	0	0
Original Code	1	0	0	0	1	0	1	1

The following block diagram shows the various steps discussed earlier.

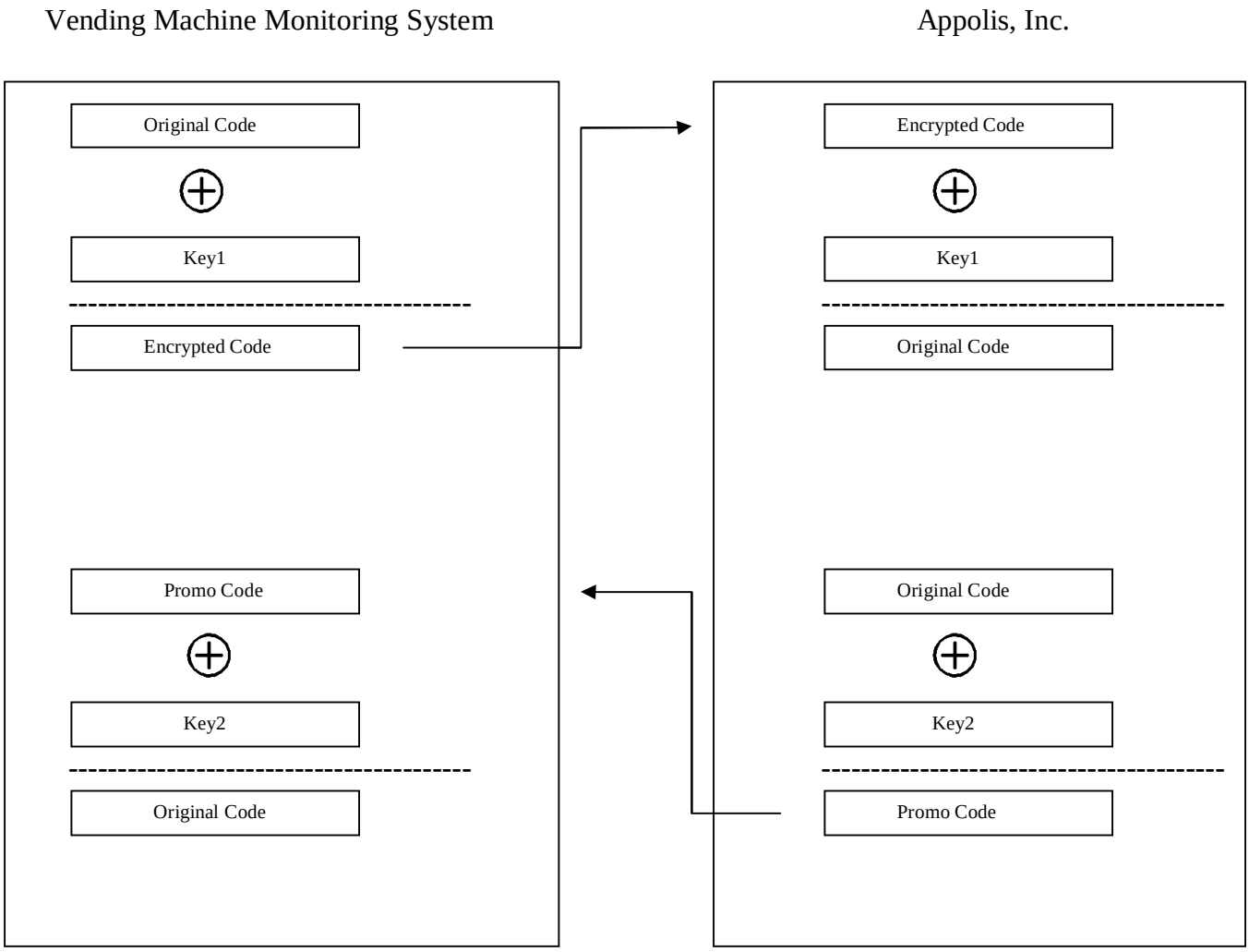


Figure 6 - Block Diagram for Encryption Algorithm

C. USER INTERFACE

V. Schematics

A. LCD & Daughter Board

Figure 7 below shows how the LCD, daughter board, and LED are connected. To connect the LCD and daughter board, we placed the daughter board face up with pin header X2 on top of the LCD face down and with the pins aligned before soldering each pin. The red LED is connected to pin 15 and 16 on the LCD. Because pins 15 and 16 on the LCD were intended for a backlight, there was a function provided for this from SJJ Micro (manufacturer of dev. board), which made this easy to use as a 5V output to flash the LED. Also, a resistor was placed in series with the LED to limit the current drawn to prevent burn out. Pin 15 is +5V, and pin 16 is the ground pin.

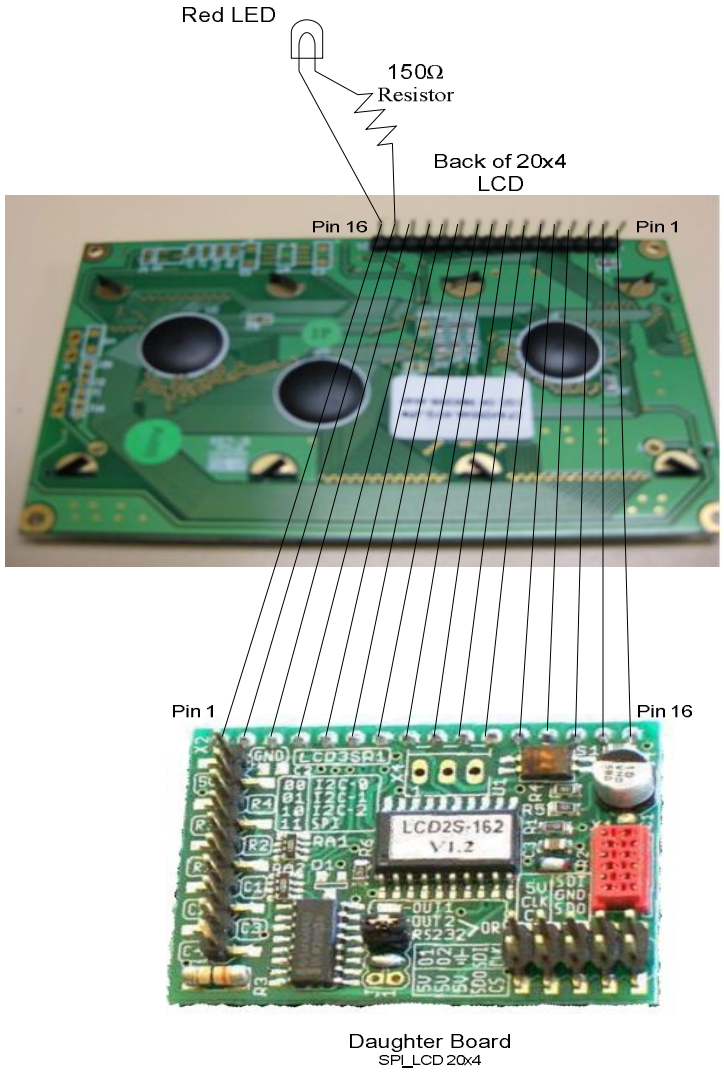


Figure 7 - Schematic of the daughter board, LCD, and LED connections

B. Keypad, Daughter Board, and Development Board

Figure 8 below shows how the development board, daughter board, and keypad are connected. Pin 10 on header X2 was not used on the daughter board, because we used a 4x3 keypad meaning there was not a fourth column (i.e. a 4 x 4 keypad) of buttons on the keypad. Also, pins 1 and 2, ground and 5V respectively, on header X2 weren't used. This was because the power and ground were already taken care of by the connection between the development board header X2 and the daughter board header X3.

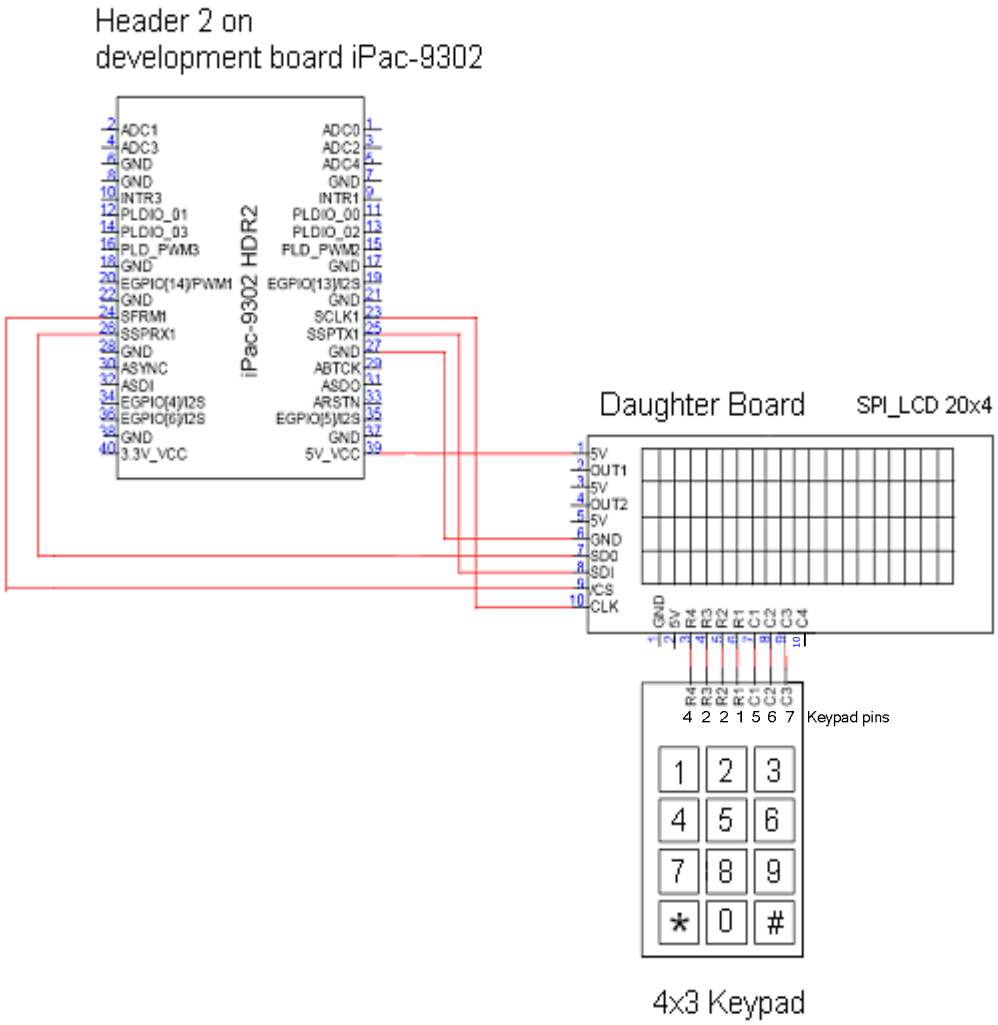


Figure 8 - Schematic of the daughter board, keypad, and development board.

C. Power Connection

Since there will not be a separate plug-in for power for the final product, we had to find power from the vending machine circuits. We were able to use the coin changer interface (P7) connector to power the development board. Pins 3 and 4 on the coin changer interface P7 were connected to header 4 pins 1 and 2 respectively, as shown below in figure 9.

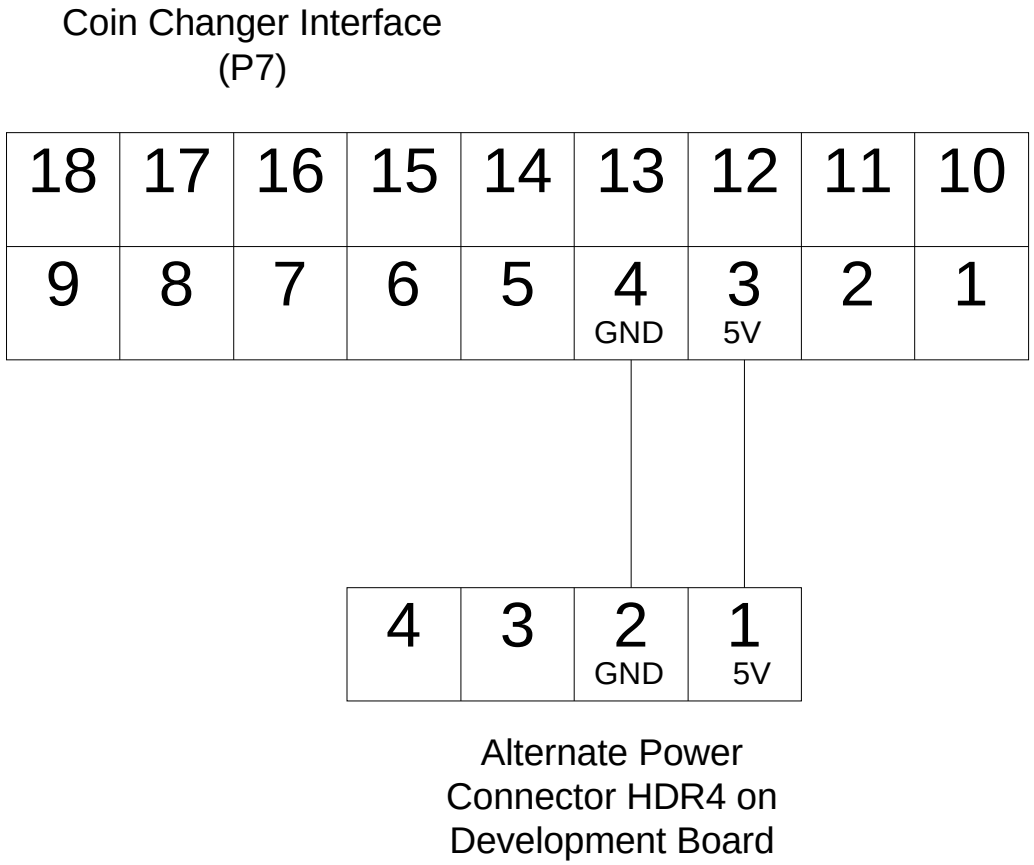


Figure 9 - Power Connections

VI. Software

DEX Serial Communication

The software for this project was written using the C# programming language and Microsoft Visual Studio 2005. The DEX communication portion required a large amount of time and testing to successfully step through the full communication. Using the windows console application saved much time as instant feedback was received to alert us if we were getting the right message back from the vending machine controller. The windows application is stored in the project folder named "DEX-windows". The DEX audit program for the ARM micro processor on the development board is located in the project folder named "DEX-ARM"

The first step using the console application was to configure the serial port to the correct baud rate, which was 9600 bits per second in this case. After this part was successfully configured, we had to get the commands correctly setup for writing to the serial port. Another issue we encountered was which data type could be sent through the serial port. Originally, we attempted to use an integer or byte to store the commands we sent to the vending machine controller. We learned that byte arrays needed to be used for sending data through the serial port.

For example:

- "*byte[] ENQ = { 0x05 };*"
- Not correct for sending through serial port: *int ENQ = 0x05;*
- Also byte ENQ = 0x05 does not work

For sending data through the serial port, byte arrays need to be used. Even if this array is one character long, this is the correct way to define a command which will be sent through the DEX port.

Once we were able to get the issues sorted out for sending commands to the machine, we sent and received data for the first time. Another item we learned to watch for was clearing the buffer (ccbuf) that we used for reading the data from the vending machine. If the buffer was not cleared, the data would be read into the start of the array, and overwriting only the bits received. This caused confusion a couple times where we thought we were receiving data in a certain stage, but only the first bit changed to "ENQ", while the rest stayed the same as the previous step.

Main Program

As shown below in figure 10, the main program flow chart included the three handshake portions. The flow charts for each of those will follow the main program. As listed in the text in figure 10, 'send "ENQ"' refers to writing that command to the serial port. Condition blocks about commands being "received" refer to receiving data from the vending machine controller through the serial port. As stated above, after ENQ is sent, if ENQ is received back, the timing was wrong so this loops back to sending ENQ again. When DLE is received, the process can proceed to the master handshake portion of the flow chart.

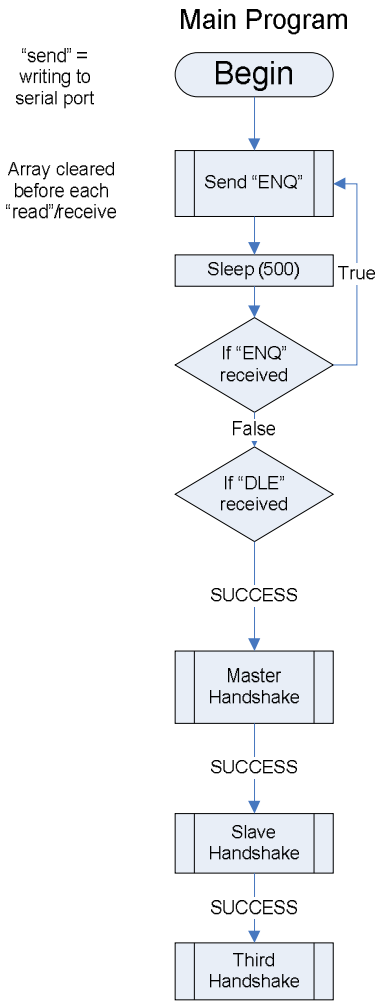


Figure 10 - Main Program flow chart

Master Handshake

Figure 11 - Screen shot from the windows console application (written in C#) used for testing the communication.

Shown above in figure 11 is the console application that was used to check progress and to know if we were getting the correct responses to progress through the handshakes. We printed each byte that was sent to the console, and preceded showed the response afterward. For the master handshake, we sent the

variables shown. After receiving “DLE” (decimal 16, hexadecimal 10) and “49” (hexadecimal 31), the master handshake was complete. Another item to note is that the console application showed the decimal equivalent of each byte received from the vending machine through the serial port. The data we received from the machine, besides the 32 control commands. We were able to use Microsoft Excel to quickly convert the decimal values we received into ASCII characters, so we could see the data we received from the machine. Included in Appendix B is the table of ASCII characters, which shows ASCII values along with hexadecimal, octal, and decimal numbers.

Shown below in figure 12 is the master handshake flow chart. The sleep times were determined from testing, although we referred to the reference C++ code from the European Vending Association. The sleep times (delays or wait times) shown, are millisecond values. We utilized the C# “Sleep” command which simplified that part. The sample code and flow charts we used were much longer, our project’s version is a simplified version which reduces the program run time.

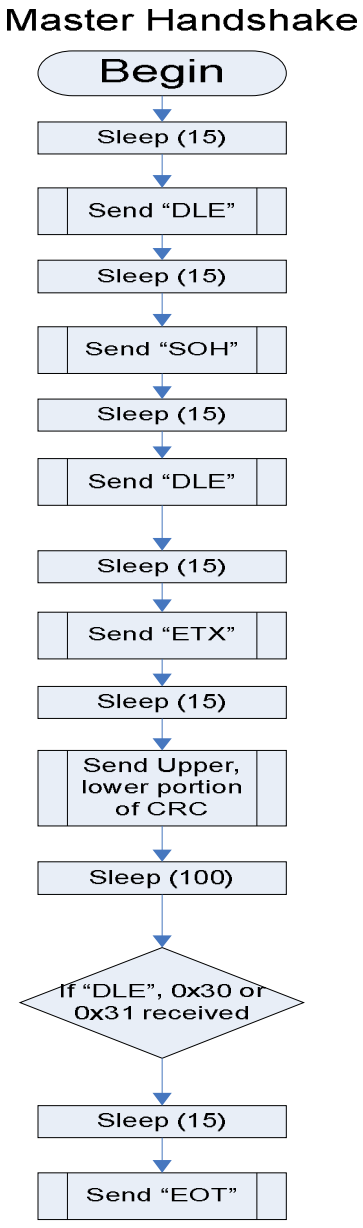


Figure 12 - Master Handshake flow chart

Slave Handshake

The flow chart for this stage is shown below in figure 13. After we received 16 and 49 from the vending machine, this moved us into the slave handshake portion of the communication. As shown below in figure 14, we sent "EOT" to the vending machine controller after the 16 and 49 were received. "EOT" stands for end of transmission, and was sent at the end of each handshake. Our software was set up to go to the alternate answer function if an ENQ (5) was received from the vending machine controller, which is the first portion shown below. Comparing line 7 (after "Received:") to line 13 (after "Received 1st:") shows how not clearing the buffer can be deceiving (the left-over values still show). Only an ENQ (5) was received after the line of (16, 1, 48, 48…). Reading the values could lead one to believe that after ETX was received, that the values (5, 1, 48, 48,…) were received. As I mention in the lessons learned section and earlier in the software section, we later learned to clear the buffer before reading so that these values do not show and cause confusion.

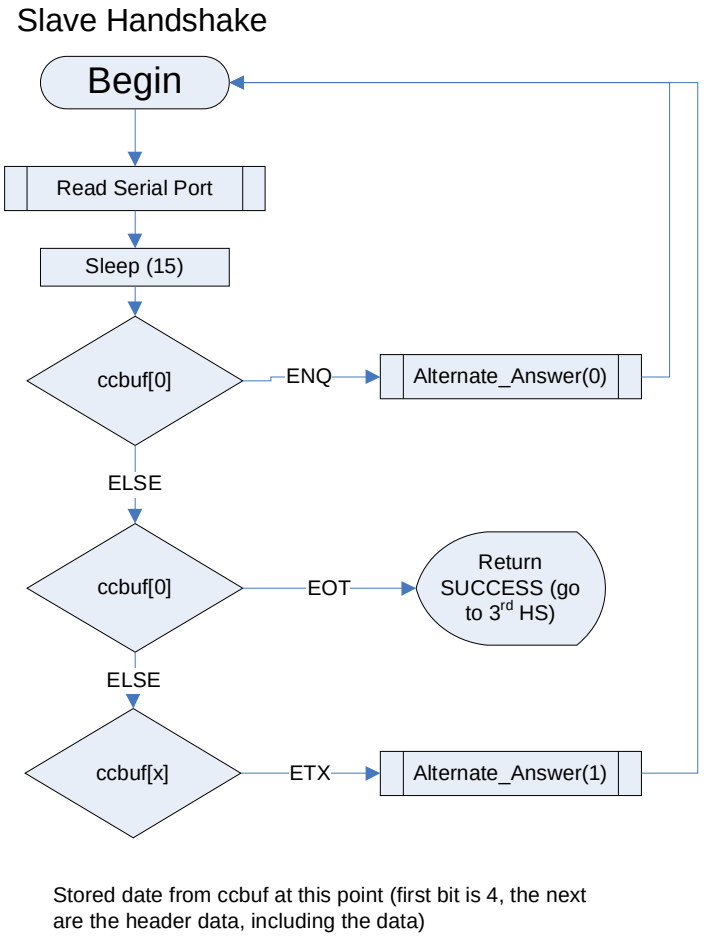


Figure 13 - Slave Handshake Flow Chart

```
Received after sending EOT:
5,0,0,0,0,
Alternate Answer: sending DLE and 0x30
AltAnswer = True

Sending DLE, 0x30 after ENQ(5) is received
Received:
16,1,48,48,54,49,53,48,50,49,48,48,48,48,82,48,49,76,48,49,16,3,188,34,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

Received ETX
Received 1st:
5,1,48,48,54,49,53,48,50,49,48,48,48,48,82,48,49,76,48,49,16,3,188,34,0,0,



Run Alternate Answer again + 0x31

Alternate Answer: sending DLE and 0x31

Alternate Answer: sending DLE and 0x30
AltAnswer = True

Sending DLE, 0x30 after ENQ(5) is received
Received:
4,5,16,2,68,88,83,42,54,49,53,48,50,49,48,48,48,48,42,86,65,42,86,49,47,49,42,49
,13,10,83,84,42,48,48,49,42,48,48,48,49,13,10,73,68,49,42,42,42,57,57,56,53,42,4
2,42,13,10,73,68,52,42,50,42,49,13,10,69,65,51,42,42,49,56,49,49,49,57,42,49,56,
48,57,13,10,16,23,38,191,0,0,0,0,0,0,0,0,0,0,Rcvd4
Success!
End of Slave Handshake!
```

Figure 14 - Windows console application portion for slave handshake.

Analysis showing bit-by-bit the meaning of the data received (shown on line 7 after "Received:"). Note that all data received will be shown on the computer as decimal values if no conversions are done.

- 4: EOT – means "end of transmission" confirmed by machine, so data transfer will follow.
- 5: ENQ – not significant in this placement, as data to follow contains more of header text
- 16: DLE: beginning of each received block
- 2: STX – start of text: means that the bytes to follow will be ASCII text.
- 68: ASCII value: D
- 88: ASCII value: X
- 83: ASCII value: S
- 42: ASCII value: *
- 54: ASCII value: 6
- 49: ASCII value: 1
- 53: ASCII value: 5

Continuing the conversions, the string reads "DXS*6150210000*VA*V1/1*1". This matches the second line of the sample DEX file, which is included in the Appendix A.

Converting the decimal values from decimal to ASCII values was necessary for our testing. For the current code in the program, we do not need to perform the conversion. We simply compare the current date to the sold-out date, so the bits can be compared in decimal form. A good example of where converting the decimal values to ASCII values is shown in the box above is located on the 7th line of the console window, after "Received:".

Third Handshake

The third handshake is the stage where the audit data is sent from the vending machine controller. The reference code we used from the EVA used more complex code for this stage, but we found that a simpler method would work for this process. Once the EOT (end of transmission) was received, we noticed that data was not received until “alternate_answer(1)”, which is a function that sends DLE and Hex 31 to the DEX port. The reference code we used started at with “alternate_answer(0)”, meaning that DLE and Hex 30 were sent. This caused the data transfer to get stuck after PA6, which as can be seen from the sample report in Appendix A, is approximately half way through the data transmission.

The final version of the code goes through the “for loop” 26 times, meaning that 26 separate text blocks are received from the DEX port. Rather than saving all the data into variables and comparing them later, our program compares the variables “on the fly”. In other words, using embedded “for” loops, the program searches for the product numbers, and then the sold out date. The sold out date is stored into an array (our program uses PA5). Each time through the loop, the program compares the audit date (today’s date) with the sold out date. Each time a sold out date occurs, the corresponding bit in the raw code is set high. For example, if products 2,3, and 5 are sold out, the raw data will be: [0,1,1,0,1,0,0,0]. Once this raw code is created, the encryption portion, covered later in this document, encrypts the code using keys, making the string that is submitted by the customer meaningless to their eyes.

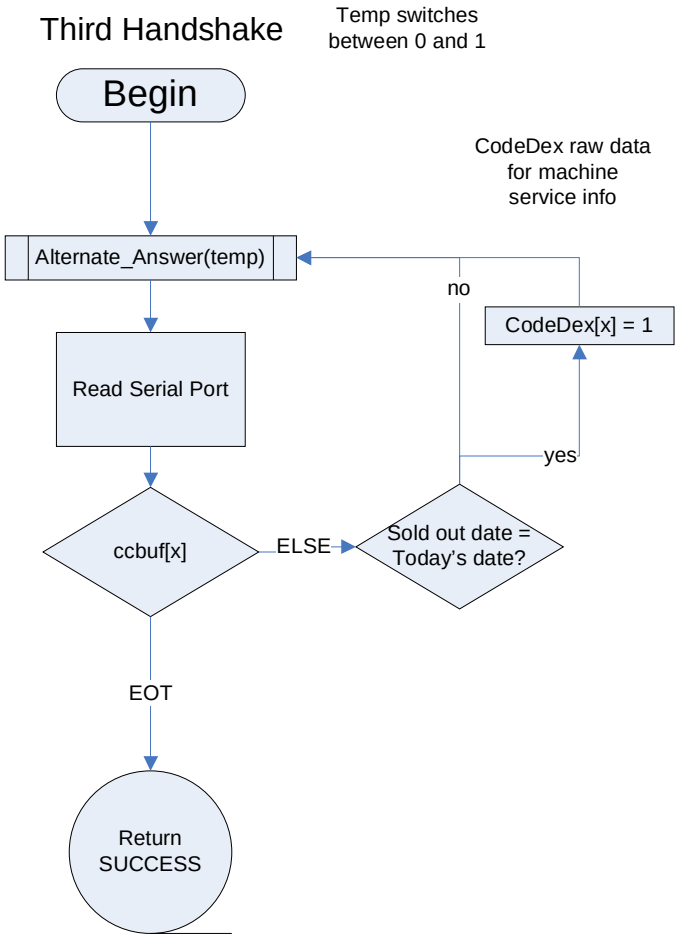


Figure 15 - Third Handshake Flow Chart

Transferring code from console application to ARM processor

We encountered a few issues that took some time to fix when transferring the working code from the computer to the ARM processor on the development board. The first issue was how the serial port was defined. See the differences between the two below. Using the ARM processor we needed to separately define the configuration settings, and then create an object for use within the function. For example, after "port1" was defined, the command "port1.Write(ENQ, 0, 1);" will send the ENQ command through the serial port to the vending machine controller. The first item passed in the "write" command is the variable being sent, the second is the start bit of the array being sent, and the third is the number of bits being sent.

ARM code:

```
static SerialPort.Configuration COM2Port_Config = new SerialPort.Configuration
(SerialPort.Serial.COM2, SerialPort.BaudRate.Baud9600, false);
public SerialPort portDex = new SerialPort(COM2Port_Config);
```

Windows App Code:

```
static private SerialPort port1 = new SerialPort("COM1", 9600, Parity.None, 8,
StopBits.One);
```

VII. Technician Troubleshooting

Problem	Parts affected	Solution
LCD does not power up	LCD	• Check the connections from LCD to iPac-9302 • Check if LCD is damaged
LCD displays black blocks	LCD	• Replace the LCD
LED does not blink	LED	• Check the connections on the LCD2S • Replace LED
Keypad does not respond	Keypad	• Check the connections on the LCD2S • Set the keypad debounce time to a lower value • Replace Keypad
DEX Communication does not complete	Serial to Audio Connector	• Secure the connector at both ends • Replace the connector
	RS232 port	• Check if the RS232 port on iPac-9302 is working
	Audio port	• Check if the Audio port on the Vending Machine is working
	Vending Machine	• Check for bottles jammed in the machine
	Software	• Cycle power for the Vending machine and iPac-9302 • Try test code with computer to machine

VIII. Project Comments

Potential Future Issues

- Machine will not vend while DEX audit occurs (coin will just pass through)
  - Program estimated run time: 20-30 seconds
- Current method of comparing dates should be improved
  - With current software overlaps could be missed (i.e. sold out before midnight, audit happens after midnight)

Project Issues

- Finding information detailing the DEX communication and standards
  - During the second semester we found a standards document which gave a complete picture of how the audit from the vending machine controller worked.

Lessons Learned

- Byte arrays must be used for sending and receiving data through the serial port (for C#)
- The serial port is defined differently for the ARM processor on the development board than the serial port on the computer that we used.
- For any buffers used for receiving text, be sure to clear the buffer prior to receiving a second time
  - If this is not done, it will appear that the same data has appeared, besides the first bit changing
- Timing with serial communication is very important
- If the LCD displays black blocks, it is defective!!!
-

Future Improvements

- Inventory Readings
- Reset button to be pushed after machine serviced
- Expand code passed to include all numbers (0-9, currently 0 and 1)
- Design PCB with ARM microcontroller and only needed components (cost savings)
- Improve outer enclosure to better protect against vandalism

Tips for Future Design Students

- When creating a project in Visual Studio, and a box shows up saying “This project is not trusted”, you need to save the project on a local hard drive (the “C” drive in most cases).
  - If a non-trusted project is used, opening the serial port will fail and give a “security exception” warning.

IX. Project Costs

Item	Unit Cost	Acquired Cost	Qty	Notes
Pepsi Machine	$ 600.00	$ -	1	Bought by Travis
Main Components				
20x4 LCD Display	$ 20.00	$ -	1	
Keypad	$ 12.26	$ 12.26	1	Digi-key part #GH5008-ND
LED	$ 2.00	$ 8.00	4	Red panel-mount, two different sizes
LCD2S	$ 37.95	$ 37.95	1	
Mechanical				
Ribbon Cable Connector	$ 2.60	$ -	2	10 pins
10-pin socket connector	$ 3.00	$ -	4	
DEX connector/adapter	$ 20.00	$ -	1	Got connectors and made cable
DEX Splitter	$ 18.00	$ 18.00	1	Not used
Enclosures				
Inner	$ 12.64	$ 12.64	1	Hammond model 1554N
Outer	$ 10.98	$ 10.98	1	Hammond model 1592
Other necessary tools				
Development kit - EDK Plus (SJJ Embedded Micro Solutions)	$ 225.00	$ 225.00	1	
C# Embedded Programming tutorial	$ 15.00	$ 15.00	1	
Totals:	$ 979.43	$ 339.83		

X. Appendix A

Example of a DEX file (ASCII characters)

DXS*6150210000*VA*V1/1*1
ST*001*0001
ID1***9985***
ID4*2*1
EA3**181201*1715
VA1*498475*4253*1650*66
CA3*2355*0*2355*0*536530*142155*43675*3507
CA4*3730*3275*42415*4070
DA2*0*0*0*0*0*0
DA3*0*0*
TA2*0*0*0*0*0
LS*0001
PA1*1*25*
PA2*554*63900*18*450
PA5*181126*1728*38
PA1*2*25*
PA2*1039*124375*17*425
PA5*181201*1628*9
PA1*3*25*
PA2*721*80675*11*275
PA5*181120*1611*9
PA1*4*25*
PA2*631*73700*6*150
PA5*181120*1611*13
PA1*5*25*
PA2*418*49175*1*25
PA5*181201*1704*12
PA1*6*25*
PA2*475*56100*11*275
PA5*181201*1706*10
PA1*7*25*
PA2*415*50550*2*50
PA5*181119*0757*16
LE*0001
LA1*2*1*1250*0*0
LA1*2*2*1250*0*0
LA1*2*3*1250*0*0
LA1*2*4*1250*0*0
LA1*2*5*1250*0*0
LA1*2*6*1250*0*0
LA1*2*7*1250*0*0
MA2*111222333*DNC-7/7
EA2*DO*98*398

EA2*CR*0*1
EA7*86*152
MA5*1*0001*1
MA5*2*0002*2
MA5*3*0004*3
MA5*4*0008*4
MA5*5*0010*5
MA5*6*0020*6
MA5*7*0040*7
MA5*ERROR*CJ6*1612*181120
MA5*ERROR*CJ2*1612*181120
MA5*ERROR*CJ3*1612*181120
MA5*ERROR*CJ4*1612*181120
MA5*ERROR*CJ5*1612*181120
MA5*ERROR*SS8*0651*181119
MA5*ERROR*SS1*0640*181119
MA5*ERROR*SS2*0650*181119
MA5*ERROR*DS*1630*181126
MA5*ERROR*DS*1630*181126
MA5*ERROR*DS*1630*181126
MA5*ERROR*DS*1630*181126
MA5*ERROR*DS*1630*181126
MA5*TUBE1**0*0*0*0
SD1*000000*
G85*618F
SE*68*0001

DXE*1*1

XI. Appendix B

Table of ASCII Characters

This table lists the ASCII characters and their decimal, octal and hexadecimal numbers. Characters which appear as r
non-printing characters appears after this table.

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

Figure 16 - ASCII table required for interpreting data received from machine.

XII. Appendix C

Development kit

The choice of the micro controller was based on the fact that the software for the project was to be coded in C#. Based on this premise, we chose the Embedded Development Kit **iPac-9302 Plus** from SJJ Embedded Micro Solutions.

-The following information is from SJJ Embedded Micro Solutions, LLC guide to the *Embedded Development Kit for the Microsoft .NET Micro Framework.*

The iPac-9302 was specially designed for the .NET MF. There are plenty of I/O options to build a variety of applications. Even though .NET MF currently doesn't support some of the ports like Ethernet, USB Host, or SD card, these features will be added over time. The board itself meets the PC/104 dimensions so you can take advantage of PC/104 power supplies and enclosures.

A majority of the port signals are on 3 header blocks, while serial connections have their own separate 10 pin headers. There are two LEDs on board. The Red LED is a boot indicator not available for program control. The Green is available for program control through the GPIO interface. There is more than enough RAM and FLASH (8MB/8MB) for application and data storage.

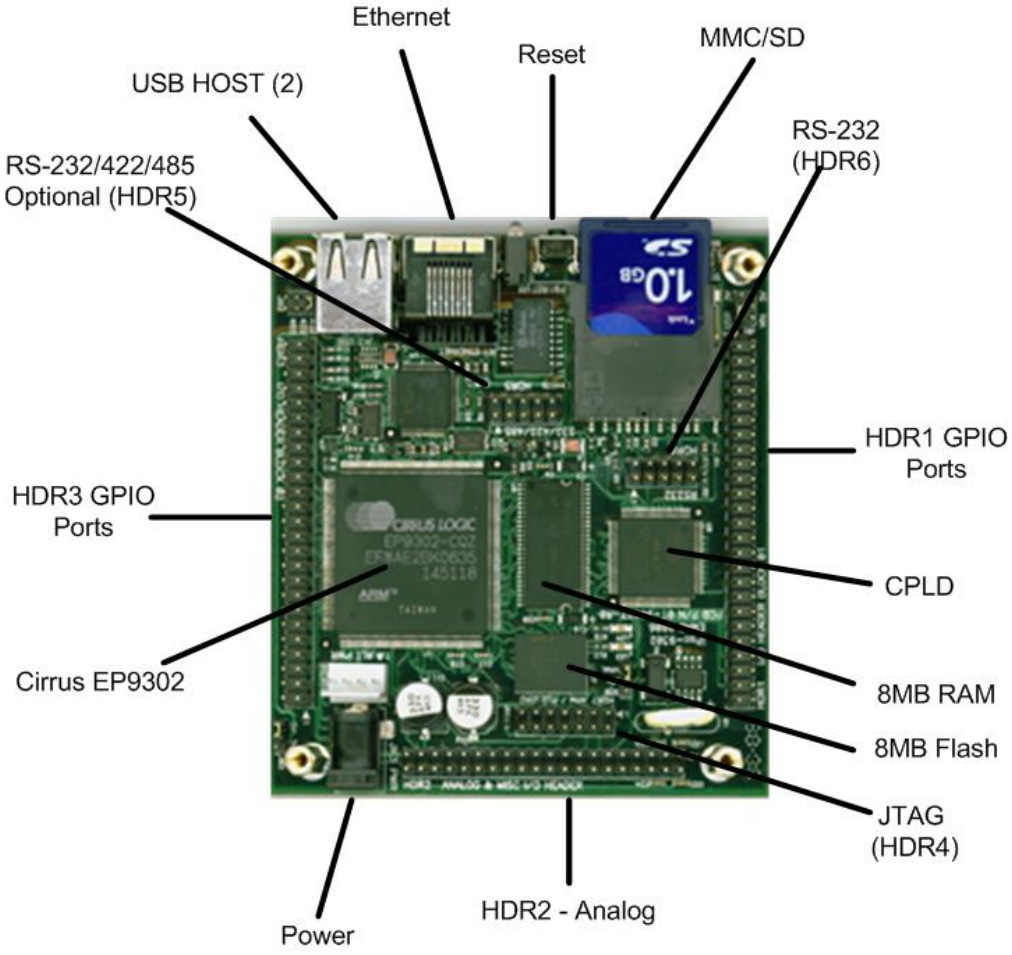


Figure 17 iPac9302 EDK

iPac Features relevant to the project

• PC/104 Dimensions of 96 mm x 90 mm (3.77" x 3.54")
• 1 RS232
• 5 channels of 12 bit A/D (0 to 3.3V)
• Internal Real time clock/calendar (no battery backup)
• A 50 pin header with 16 processor GPIO lines and 8 PLD output GPIO lines.
• A 50 pin header with 8 PLD 5 volt tolerant GPIO lines, 8 PLD output GPIO lines, and 8 PLD High Drive output GPIO lines.
• 2 PWM I/O lines
• SPI / AC97 / I2S options
• External Reset Button provision and red & green Status LEDs
• 8 MB of External Flash
• 8 MB External SDRAM
• External SPI Battery backed clock/calendar
• 16 / 32 / 64 MB meg of Flash PC28F256P30
• 32 / 64 MB of SDRAM MT48LC16M16 / 48LC32M16
• 1 RS232/422/485 serial port

Boot Loader

There is a boot loader called TinyBooter that is on the flash chip. The boot loader will help with upgrading CLR in the future. The system takes about 7 second to launch an application. The delay allows for upgrading the CLR via TinyBooter. During a normal boot operation, the iPac-9302's boot hardware will detect the boot loader, TinyBooter, and execute it. TinyBooter will do basic hardware configuration and then listen for commands on the debug serial port for 5 seconds. If it receives commands, it will respond.

One such command is to flash a new version of the CLR, and another is to erase the managed code and data areas. This will be discussed in more detail, later. Typically, no commands are sent to TinyBooter, so it passes control to the CLR. CLR looks for a managed code application to run. If no managed code application is found, CLR, listens to the debug serial port for commands, such as a connect command from Visual Studio. If a managed code application is found, it loads it and passes control to it.

Flash Memory Map

TinyBooter, CLR, and managed code applications and data are all stored in flash memory. The TinyBooter image is stored in the "Bootstrap" area of flash memory. When TinyBooter is run, a decompression utility runs out of flash that decompresses the compressed TinyBooter image to RAM and jumps to the decompressed image. At this point, TinyBooter runs from RAM.

CLR and its configuration data are stored in the "Configuration" and "Code" sections of flash memory. When CLR is running, it has to be able to write to flash memory. When CLR is writing to flash memory, it cannot be executing its instructions from flash memory at the same time; therefore, when control is first passed from TinyBooter to CLR, CLR relocates certain of its routines from flash memory to RAM, where they will accessed during normal execution. Routines such as those that write to flash memory are relocated to RAM and executed from RAM during normal CLR operation. With the exception of routines like those, CLR runs from FLASH. Managed code modules and their associated data are stored in the

"Deployment" section and "Storage" section of flash memory respectively. Managed code modules and data almost exclusively flash memory entities.

SPI Interface

The Serial Peripheral Interface Bus (SPI) bus is a synchronous serial data link standard that operates in full duplex mode. It is sometimes known as a "four wire" bus because it has four basic signals – chip select, clock, transmit, and receive.
Unlike RS232 communications, SPI allows more than one device to exist on the bus in a host/client configuration. The iPac-9302's SPI port acts like the master-host and there can be several client devices on the bus. GPIOs combined with the SPI port's chip select, can be used to create unique slave select signals for each slave device that is connected to the SPI bus. This provides the ability to connect several SPI slave devices to the iPac-9302's SPI port.

The naming and phasing of the SPI signals vary from vendor to vendor. The iPac-9302 uses the following signals:
• SCLK1 – SPI Serial (Bit) Clock
• SSPTX1 – SPI Serial Output
• SSPRX1 – SPI Serial Input
• SFRM1 – SPI Frame Clock or Chip Select

Only one client device can talk to the master-host. All SPI devices are tri-stated when not selected. Only one client device can be selected at a time, so there are no conflicts on the bus.

Communications starts with the master-host configuring a clock frequency. The frequency must be less than the maximum frequency that the client device can support, and the frame-to-frame time must be greater than the minimum frame-to-frame time supported by the client device. The master-host then signals a chip select low to enable the desired client device.

Then on each SPI clock cycle, a full duplex data transmission occurs. The master-host sends data on the SSPTX1 (SPI Serial Output) line, and simultaneously with each SPI clock cycle, data is sent from the client to the master-host on the SSPRX1 (SPI Serial Input) line. The data sent out on the SSPTX1 line is sent out through a shift register in the SPI controller. The client then reads the data sent on the SSPTX1 line. The client can send data back on the SSPRX1 line, but only through a write/read operation initiated and controlled by the master-host. The master-host then reads the data on the SSPRX1 (SPI Serial Input) by shifting the SSPRX1 data in and doing a serial to parallel conversion.

Serial LCD Daughter board

-The following information is from Modtronix Engineering's guide *LCD2S Revision 1, Firmware V1.20.*

We use a Serial LCD Daughter board to connect the LCD and Keypad to the iPac9302. The daughter board is called LCD2S and can be connected to a standard 20x4-character display module that supports 4-bit mode. The LCD2S-204 has a high-speed SPI/I2C serial bus. For convenience, the I2C and SPI signals are available via a 6-pin Micro-MaTch type connector, and a standard 2.54mm, 2x5-row pin header. A keypad with up to 16 buttons (4 rows by 4 columns) can be added to the display via a standard 10-pin, 2.54mm header. When using the I2C serial bus, an interrupt line will be activated when a key is pressed, informing the host that there are key data to be read. Alternatively, the LCD2S-204 can be polled to see if it has any pending keypad data.

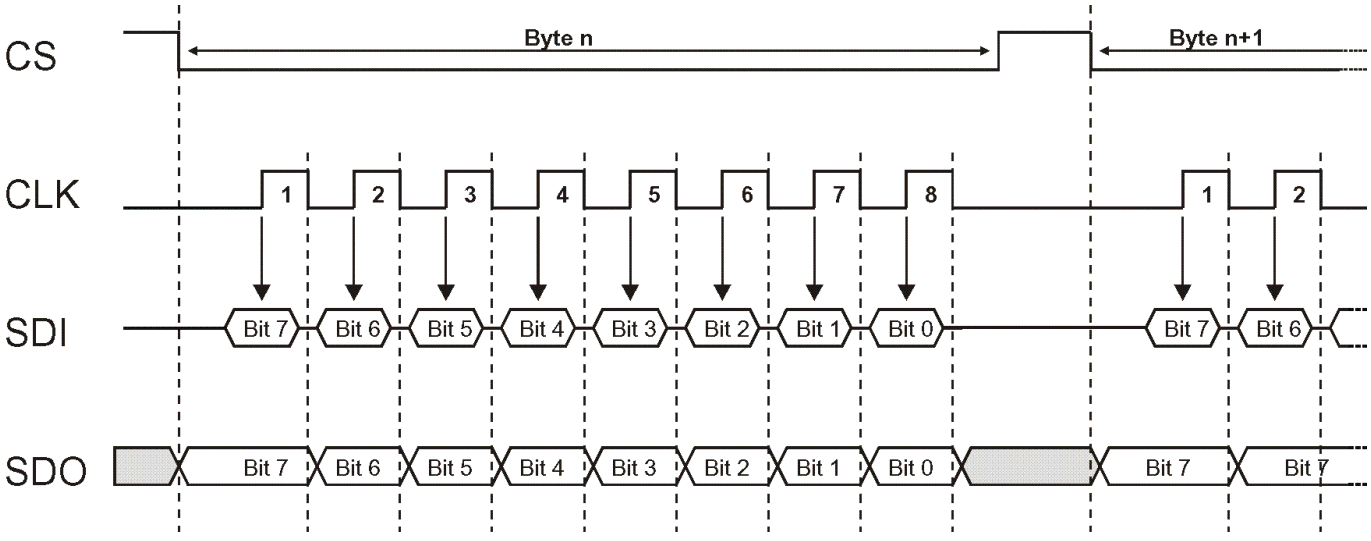
The board has 5 user-configurable, general-purpose inputs/outputs. Two of them can deliver up to 1000mA each, and have protection circuitry for driving relays.

LCD2S Features relevant to the project

• Can be controlled via SPI or I2C bus
• Assembled with 2x16 or 4x20 line character displays, with or without back lighting
• Two high current, open collector outputs (OUT1 and OUT2), with output current of 1000mA each
• 3 general purpose input/output pins (GPIO1, GPIO2 and GPIO3) that can be used for:
• 2 Analog inputs with 10 bit resolution and 1 digital I/O
• 3 Digital I/O lines that can be used for inputs or outputs (1k output impedance).
• Keypad encoder for a keypad up to 16 keys (4 rows by 4 columns)
• Keypad has a configurable button repeat delay and repeat rate
• User configurable start up screen
• 80 Byte buffer for messages received via serial interface
• Large number of LCD commands for moving cursor, moving display and more
• Up to 8 custom characters can be defined
• 6 Pin Micro Match connector with I2C, SPI and power signals
• 10 Pin (10 pins x 1 row), 2.54mm connector for connecting a keypad of up to 4 rows by 4 columns
• 10 Pin (5 pins x 2 rows), 2.54mm connector with user outputs, I2C, SPI and power signals
• DIP switch for setting serial mode and I2C address
• Low current consumption - without LCD backlight is about 4mA

Communication via SPI

This communication method is selected by setting the DIP switch to 11 (both switches are in the on position). SPI mode 0 is used. This method uses 4 signals for communication, CS, CLK, SDI and SDO. The SPI CS signal is used to enable communication with the LCD2S. Whenever the CS signal goes low, the SPI port on the LCD2S becomes active. In this state the SDO line will be an output, and the SDI and CLK lines will be inputs. Applying a clock signal to the CLK input will clock data into the LCD2S via the SDI line, and data out via the SDO line.
The figure below shows the timing diagram for this mode of communication.



When the CS signal is set high, the SPI port is disabled. In this state, the SDO line is tri-stated and the SDI input is disabled. The
LCD2S will not respond to any data on the SPI bus, and will also not interfere with data been send between any other nodes that potentially share the SPI bus with it.

Keypad

A keypad of up to 16 keys (4 rows by 4 columns) can be connected to the LCD2S. All keys are filtered via a configurable software debounce filter. Each time a key is pressed, a key code from 'a' to 'f' is added to the 16 byte keypad FIFO buffer. See Figure 8 for a wiring diagram and key codes. The keypad buffer can be read via the "*Read Keypad Data*" command.

In I2C mode there is an I2C interrupt line that will be pulled low by the LCD2S if it has pending data. Currently pending data will always be keypad data. If the I2C interrupt line is not used, the LCD2S has to be polled each couple of ms to see if it has any keypad data available. This is done by reading the status byte with the "*Read Device Status Byte*" command. The status byte will indicate if there is any pending keypad data.

Keypad with 12 Keys

In this mode, a keypad of up to 4 rows by 3 columns can to be used. To configure the LCD2S for a 12 key keypad, the following must be done:
– The keypad must be configured for a 4x3 keypad via the "*Configure Keypad and IO*" command. This command will also disable the GPIO1, GPIO2 and GPIO3 general purpose inputs/outputs.
– The GPIO1, GPIO2 and GPIO3 general purpose inputs/outputs are not available when using a 12 key keypad! The OUT1 output is available.
– The keypad must be connected to R1 to R4 (rows 1-4) and C1 to C3 (column 1-3) of the X2 connector.

XIII. Appendix D

LCD DISPLAY COMMANDS

-The following information is from Modtronix Engineering's guide *LCD2S Revision 1, Firmware V1.20*

Write Parsed String
Command: 0x80, "String"
Writes the given string to the LCD. All characters are mapped to the Character Set shown in .
The string is parsed for escape sequence characters that perform the following actions:
– 0x0a ('\n') = Go to beginning of next line.
– 0x0c ('\f') = Clear display and go to beginning of first line.
– 0x0d ('\r') = Go to beginning of first line
– 0x08 ('\b') = Cursor left
– 0x09 ('\t') = Cursor Right

Set Startup Screen
Command: 0x90, [line], "String"
Sets a single line of the startup screen. The startup screen is displayed when the LCD is switched on.

Backlight On
Command: 0x28
Switches the display on. The display brightness will be the value last set with the "Set Backlight Brightness" command.

Backlight Off
Command: 0x20
Switches the display off. The display brightness will not be modified, and will be restored the next time the "Display On" command is issued.

Cursor moves forward
Command: 0x09
Causes the cursor (or display) to move one position forward each time a character is written to the display. The default mode of the
LCD2S is for the cursor to move forward.

Cursor moves backwards
Command: 0x01

Move Cursor Right
Command: 0x83
Moves the cursor one position forward (right). This command does not change the contents of the Display RAM! When shifting the cursor right along the first line, it will move to the second line when passing the 40th character of line 1. The next character written to the LCD will be at the new location of the cursor.

Move Cursor Left
Command: 0x84

Set Cursor Position
Command: 0x8A, [row], [column]
Moves the cursor to the given row and column location.

Clear Display
Command: 0x8C
This command sets the cursor position to the first character of row 1 and writes space characters (0x20) to all Display RAM locations.

Keypad Commands

Set Keypad Debounce Time
Command: 0xE1, [value] Default: 3 = 24ms
Sets the keypad debounce time. Debounce Time = value x 8ms

Read Keypad Data
Command I2C: 0xD1, <read key>
Command SPI: 0xD1, 0x00, <read key>
Reads the next byte from the keypad buffer. When executing this command via the SPI port, a dummy byte must be sent after the 0xD1 command! If there is no key data in the keypad buffer, 0 is returned. If there is key data in the keypad buffer, a key code from 'a' to 'f' is returned, depending on what key was pressed.

SPI Controller Commands

Each of the commands discussed above are byte commands to control the LCD display. All of the commands must be proceed by a 0xF5, which provides a SYNC character to the controller to let it know a command is coming.

If we simply want to write text to the first line of the LCD display, we first need to position the cursor to line 1 and the first column of that line.
The command to set cursor position is 0x8A followed by the Row number (1 or 2) and a Column number (1-16).
If we want to set the cursor to line 1 character 0, we would send the following:

0xF5, 0x8A, 0x01, 0x01

With the cursor positioned, we can write a string of data to the LCD. The 0x80 command followed by the string will write the data to the LCD.

0xF5, 0x80, “String”

Since SPI.Write and SPI.WriteRead are expecting Byte[] or Unint16[], the string itself will need to be reconfigured for output. Since the commands for the LCD controller are a byte in length, we will use a UTF8 Encoding to convert a string to byte characters.

SPI LCD LIBRARY

Configure SPI settings and create an instance of the SPI port

```
static SPI.Configuration mySPIPortSettings = new SPI.Configuration(Pins.GPIO_NONE,
false, 0, 0, false, true, 300, SPI.SPI_module.SPI1);
SPI mySPIPort = new SPI(mySPIPortSettings);
```

The first line sets up our SPI port communication settings. Per our schematic, there is no ORing of GPIO the chip-select so GPIO_NONE is set for the chip select. The second argument set the chip select active state, so false is required for the negative going chip select of the SPI LCD controller. We are not using a GPIO chip select OR'd with the SPI frame select signal, so the next two parameters, chip select setup time and chip select hold time are not used and are set to 0's. The 5th argument is the clock idle state. Since the clock idle state is 0, we set this parameter to false. Next is the clock edge that the data is read on, and that is the positive clock edge, so we set that parameter to true. Next is the clock frequency in KHz which is set to 300 KHz. The last parameter is the actual SPI module itself, and the iPac-9302 only has one supported. Using the configuration settings, the next line creates a new instance of the SPI port.

Send the basic commands to write our string to the display

```
// LCD send byte string to lines preamble
byte[] bWriteLCDLine1Preamble = new byte[] { 0xF5, 0x8A, 0x01, 0x01, 0xF5, 0x80 };
```

The above line creates a byte array with the basic commands to set the cursor position and execute the command to write the string. This information needs to precede the string we are sending each time. Now, add the following line, which is our string to be displayed.

```
// Unicode literal strings
string sDisplayString1 = "Vending Machine Controller";
```

In order for the string to be sent, we need to convert the Unicode string to a byte array and then merge the two byte arrays together so they can be sent. Remember that all strings in .NET MF are UTF-8 Unicode, so we will use some encoding to get the bytes. Add the following lines of code:

```
// Convert string literals to byte arrays
byte[] bDisplayString1 = new byte[sDisplayString1.Length];
// convert literal unicode strings to byte arrays
System.Text.UTF8Encoding Encoding = new System.Text.UTF8Encoding();
bDisplayString1 = Encoding.GetBytes(sDisplayString1);
// Construct the final message byte array
byte[] bWriteMessageLine1 = new byte[bWriteLCDLine1Preamble.Length +
bDisplayString1.Length];
//copy the preamble for line 1 first followed by the string1 starting at the
//end of the pre-able (using the length)
bWriteLCDLine1Preamble.CopyTo(bWriteMessageLine1, 0);
```

```
bDisplayString1.CopyTo(bWriteMessageLine1, bWriteLCDLine1Preamble.Length);
```

Finally write the message to the SPI LCD:

```
Thread.Sleep(1000);
mySPIPort.Write(bWriteMessageLine1);
Thread.Sleep(6);
```

Extending the SPI LCD Library for 20x4 Display

After designing a library of commands for the 16x2 LCD, we can reuse the code for a 20x4 display by adding a few statements.

```
//Extend the SPI LCD driver to manage the 4-line x 20 character display
   public class SPI_LCD_4LDriver : SPI_LCDDriver
   {
      // Override the line length of the display
      protected static new int ilineLength = 20;

      private const int iLineIndex = 2;
      protected byte[] bSetCursorPreamble = new byte[] { 0xF5, 0x8A };
      protected byte[] bWriteAtCursorPreamble = new byte[] { 0xF5, 0x80 };

      public byte[] LCD4LLineNoWrap(string sDisplayString, byte bLine)
      {
         // Construct the display lines
         //Size the array to manage write preamble, 1 blank char for selector indicator, and actual string
         byte[] bWriteMessageLine = new byte[bWriteLCDLine1Preamble.Length +
sDisplayString.Length + 1];

         //Truncate string to size of display
         string sTrimString = "";
         for (int i = 0; (i < ilineLength) && (i < sDisplayString.Length); i++)
         {
            sTrimString += sDisplayString[i];
         }

         switch (bLine)
         {
            case 1:
            {
               bWriteMessageLine = base.LCDLine1(sTrimString);
               break;
            }
            case 2:
            {
               bWriteMessageLine = base.LCDLine2(sTrimString);
```

```
            break;
        }
        case 3:
        case 4:
        {
            bWriteMessageLine = base.LCDLine1(sTrimString);
            // Overwrite line index
            bWriteMessageLine[iLineIndex] = bLine;
            break;
        }
        default:
        {
            bWriteMessageLine = base.LCDLine1NoWrap("LCD Line Index Error");
            break;
        }
    }

    return bWriteMessageLine;
}

public byte[] WriteByteAtCursor(byte bChar)
{
    //Write a single byte to the current cursor location
    byte[] bWriteMsg = new byte[bWriteAtCursorPreamble.Length + 1];
    bWriteAtCursorPreamble.CopyTo(bWriteMsg, 0);
    bWriteMsg[bWriteAtCursorPreamble.Length] = bChar;
    return bWriteMsg;
}

public byte[] WriteStringAtCursor(string sMsg)
{
    // Convert string literals to byte arrays
    byte[] bMsg = new byte[sMsg.Length];

    // convert literal unicode strings to byte arrays
    System.Text.UTF8Encoding Encoding = new System.Text.UTF8Encoding();
    bMsg = Encoding.GetBytes(sMsg);

    byte[] bWriteMsg = new byte[bWriteAtCursorPreamble.Length + sMsg.Length];
    bWriteAtCursorPreamble.CopyTo(bWriteMsg, 0);
    bMsg.CopyTo(bWriteMsg, bWriteAtCursorPreamble.Length);

    return bWriteMsg;
}

public byte[] WriteByteChar(byte bChar, byte bDisplayLine, byte bColumn)
{
    //Write a one-byte character to any line or column - Note: no range checking
    byte[] bWriteSelectorMsg = new byte[bWriteLCDLine1Preamble.Length + 1];
```

```
        bWriteLCDLine1Preamble.CopyTo(bWriteSelectorMsg, 0);
        bWriteSelectorMsg[bWriteSelectorMsg.Length - 1] = bChar;
        bWriteSelectorMsg[2] = bDisplayLine;
        bWriteSelectorMsg[3] = bColumn;
        return bWriteSelectorMsg;
    }
    public byte[] WriteByteChar(byte bChar, byte bDisplayLine)
    {
        //Overload: Write a one-byte character to the first column of any line
        byte[] bWriteSelectorMsg = new byte[bWriteLCDLine1Preamble.Length + 1];
        bWriteLCDLine1Preamble.CopyTo(bWriteSelectorMsg, 0);
        bWriteSelectorMsg[bWriteSelectorMsg.Length - 1] = bChar;
        bWriteSelectorMsg[2] = bDisplayLine;
        return bWriteSelectorMsg;
    }

    public byte[] SetCursor(byte bLine, byte bColumn)
    {
        byte[] bWriteCursorMsg = new byte[bSetCursorPreamble.Length + 2];
        bSetCursorPreamble.CopyTo(bWriteCursorMsg, 0);
        bWriteCursorMsg[bSetCursorPreamble.Length] = bLine;
        bWriteCursorMsg[bSetCursorPreamble.Length + 1] = bColumn;
        return bWriteCursorMsg;
    }
}
```

Read from the keypad

As discussed earlier, the daughter board is also connected to the keypad. So we can design a method for reading through the keypad and storing it in the buffer provided in the LCD2S.

```
public UInt32 Code(Boolean bEcho)
    {
        UInt32 iReturnLockCode = 0;
        byte[] bKeyPad;
        bKeyPad = new byte[bMaxNumLockCodeDigits + 1];
        int idx = 0;

        //Clear keypad of any extra entries
        ClearKey();

        while (true)
        {
            bKeyPad[idx] = ReadKey();
            if ((bKeyPad[idx] >= 0x30) && (bKeyPad[idx] <= 0x39))
            {
                if (idx < bMaxNumLockCodeDigits)
                {
```

```
                if (bEcho)
                {
                   mySPIPort.Write(mySPILCD.WriteByteAtCursor(bKeyPad[idx]));
                }
                else
                {
                   mySPIPort.Write(mySPILCD.WriteByteAtCursor(bAsterisk));
                }
                Thread.Sleep(6);

                //Store actual number not ASCII digit and increment idx
                bKeyPad[idx] &= 0x0F;
                idx++;
             }
             else
             {
                //Trying to add too many digits; don't store; don't increment idx; don't advance cursor
                mySPIPort.Write(mySPILCD.WriteByteAtCursor(bExclamation));
                Thread.Sleep(6);
                mySPIPort.Write(mySPILCD.LCDMoveCursorLeft());
             }
          }
          else if ((bKeyPad[idx] == (byte)DispScreen.bCancel) || (bKeyPad[idx] ==
(byte)DispScreen.bEnter))
          {
             break;
          }
          else if (bKeyPad[idx] == bBackSpace)
          {
             if (idx > 0)
             {
                //First clear any character that might be at the cursor location
                 mySPIPort.Write(mySPILCD.WriteByteAtCursor(bSpace));
                Thread.Sleep(6);
                mySPIPort.Write(mySPILCD.LCDMoveCursorLeft());
                Thread.Sleep(6);

                //Backup cursor and idx
                mySPIPort.Write(mySPILCD.LCDMoveCursorLeft());
                Thread.Sleep(6);
                idx--;

                //Clear new cursor location
                 mySPIPort.Write(mySPILCD.WriteByteAtCursor(bSpace));
                Thread.Sleep(6);
                mySPIPort.Write(mySPILCD.LCDMoveCursorLeft());
             }
          }
          else
```

```
            {
                //Trying to add non-numeric character; don't store; don't increment idx; don't advance cursor
                mySPIPort.Write(mySPILCD.WriteByteAtCursor(bQuestion));
                Thread.Sleep(6);
                mySPIPort.Write(mySPILCD.LCDMoveCursorLeft());
            }
        }

        if (bKeyPad[idx] == (byte)DispScreen.bEnter)
        {
            idx--;
            for (UInt32 uiPwrTen = 1; idx >= 0; idx--)
            {
                iReturnLockCode += bKeyPad[idx] * uiPwrTen;
                uiPwrTen *= 10;
            }
        }

        return iReturnLockCode;
    }


public byte ReadKey()
    {
        byte bKeyChar = 0x00;

        do
        {
            mySPIPort.WriteRead(SPIWriteKeyPadBuf, SPIReadKeyPadBuf);

            //Check for a valid keypad character
            if ((SPIReadKeyPadBuf[3] >= 0x61) && (SPIReadKeyPadBuf[3] <= 0x70))
            {
                switch (SPIReadKeyPadBuf[3])
                {
                    case 0x61:
                        {
                            bKeyChar = 0x31;
                            break;
                        }
                    case 0x62:
                        {
                            bKeyChar = 0x32;
                            break;
                        }
                    case 0x63:
                        {
                            bKeyChar = 0x33;
                            break;
```

```
    }
case 0x64:
    {
        bKeyChar = 0x0D;    //Enter
        break;
    }
case 0x65:
    {
        bKeyChar = 0x34;
        break;
    }
case 0x66:
    {
        bKeyChar = 0x35;
        break;
    }
case 0x67:
    {
        bKeyChar = 0x36;
        break;
    }
case 0x68:
    {
        bKeyChar = 0xC5;    //Up arrow
        break;
    }
case 0x69:
    {
        bKeyChar = 0x37;
        break;
    }
case 0x6A:
    {
        bKeyChar = 0x38;
        break;
    }
case 0x6B:
    {
        bKeyChar = 0x39;
        break;
    }
case 0x6C:
    {
        bKeyChar = 0xC6;    //Down arrow
        break;
    }
case 0x6D:
    {
        bKeyChar = 0x18;    //Cancel
```

```
                    break;
                }
            case 0x6E:
                {
                    bKeyChar = 0x30;
                    break;
                }
            case 0x6F:
                {
                    bKeyChar = 0xC8;    //Left arrow
                    break;
                }
            case 0x70:
                {
                    bKeyChar = 0xC7;    //Right arrow
                    break;
                }
            default:
                {
                    bKeyChar = 0x00;
                    break;
                }
        }
    }

    if (bKeyChar == 0x00)
    {
        //If no char, delay 1/4 second before next read
        Thread.Sleep(250);
    }

} while (bKeyChar == 0x00);

return bKeyChar;
  }
}
```

XIV. Appendix E

Encryption

After going through the DEX Communication, if an inventory emergency is detected, the inventory status message is generated. Then the method appRun() is invoked passing the two parameters viz. MachineID and CodeDex.

The method appRun() goes through the various encryption steps and the verification of the PromoCode.

```
public class App
{

 DispScreen myappDispScreen = new DispScreen();
 Dex myDex2 = new Dex();

 int i,j;
 int Coin = 0;
 int Bill = 0;
 int InvEmergency = 0;
 int Valid = 0;


 int[] Code2 = { 0, 0, 0, 0, 0, 0, 0, 0 };
 int[] SecretCode = { 0, 0, 0, 0, 0, 0, 0, 0 };
 int[] PromoCode = { 0, 0, 0, 0, 0, 0, 0, 0 };
 int[] Key1 = { 1, 1, 0, 0, 1, 1, 0, 0 };
 int[] Key2 = { 1, 0, 1, 0, 1, 0, 1, 0 };


public void appRun(byte[] MachID, byte[] Code1)
{

        UInt32 uiLockCode = 0;
        while (true)
        {

                myappDispScreen.ShowMainScreen();
                myappDispScreen.ShowMainScreen2(MachID);


         ////////////////////////////////////Screen2/////////////////////////////////////


                for (i = 0; i < 8; i++)
                {
                    if (Code1[i] == Key1[i])
                    {
```

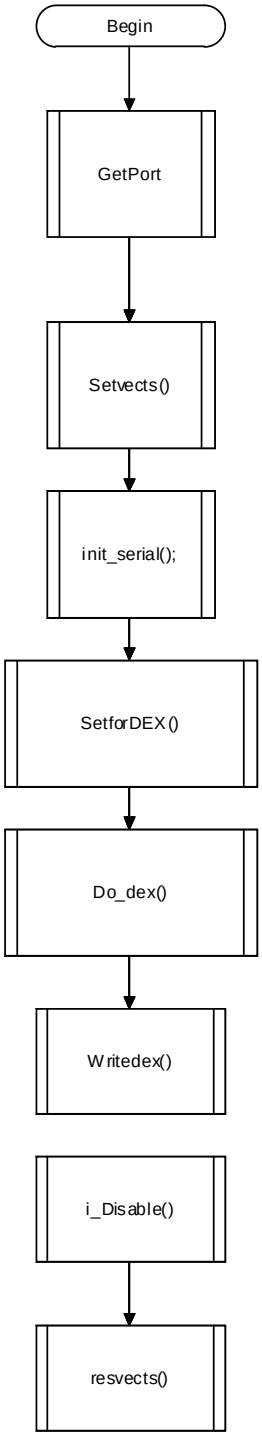
```
            SecretCode[i] = 1;
        }
        else
        {
            SecretCode[i] = 0;
        }
    }

    myappDispScreen.ShowMainScreen3(SecretCode);

    //Get keypad input
    uiLockCode = myappDispScreen.GetLockCode(bNoEcho);

////////////////////////////////////////Screen3////////////////////////////////////////////


    j = 7;
    while (uiLockCode != 0)
    {
       PromoCode[j] = (int)(uiLockCode) % 10;
       uiLockCode = uiLockCode / 10;
       j--;
    }




    //Compare the code with the original one
    for (i = 0; i < 8; i++)
    {

       if (PromoCode[i] == Key2[i])
       {
          Code2[i] = 1;
       }
       else
       {
          Code2[i] = 0;
       }
    }
    Valid = 0;

    for (i = 0; i < 8; i++)
    {
       if (Code1[i] == Code2[i])
```

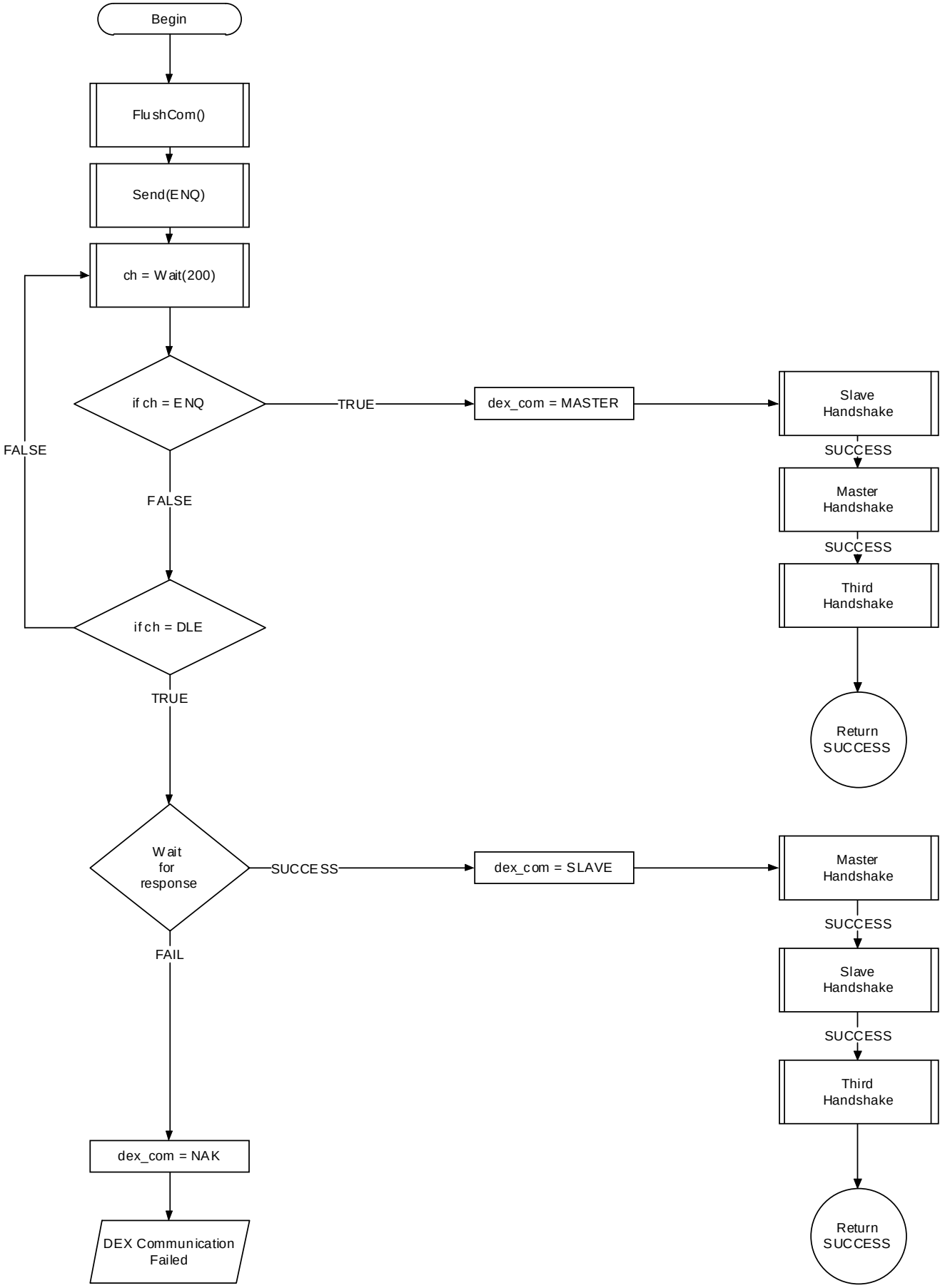
```
                {
                    Valid = Valid + 1;
                 }

             }
             if (Valid == 8)
             {
                myappDispScreen.ShowMainScreen4();
             }
             else if(Valid != 8)
             {
                myappDispScreen.ShowMainScreen5();
             }

        }
}
}
```

XV. Appendix F

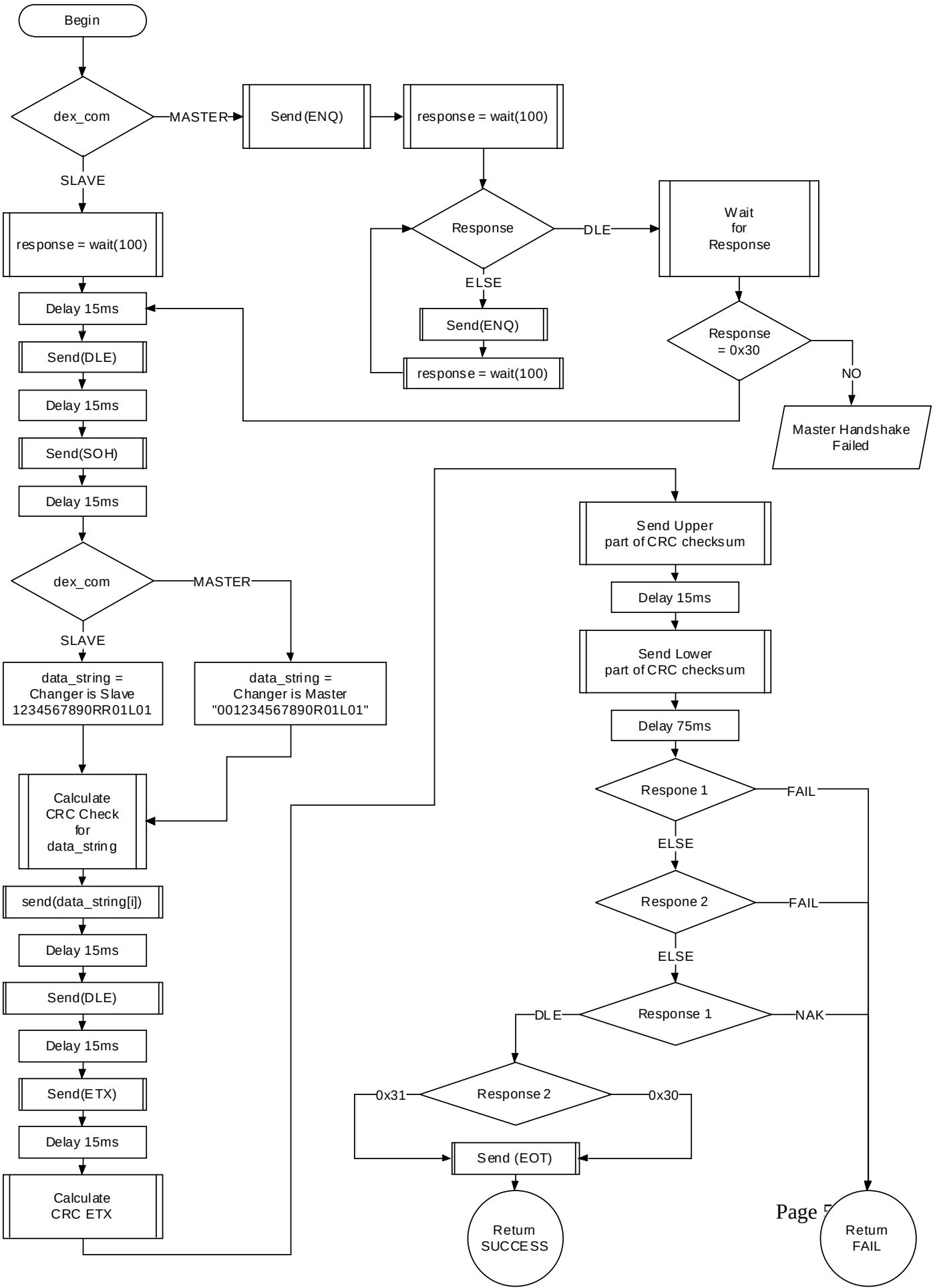
The following are DEX flow charts that come from the European Vending Association (EVA). These were used for the DEX communication.

Main Program Function

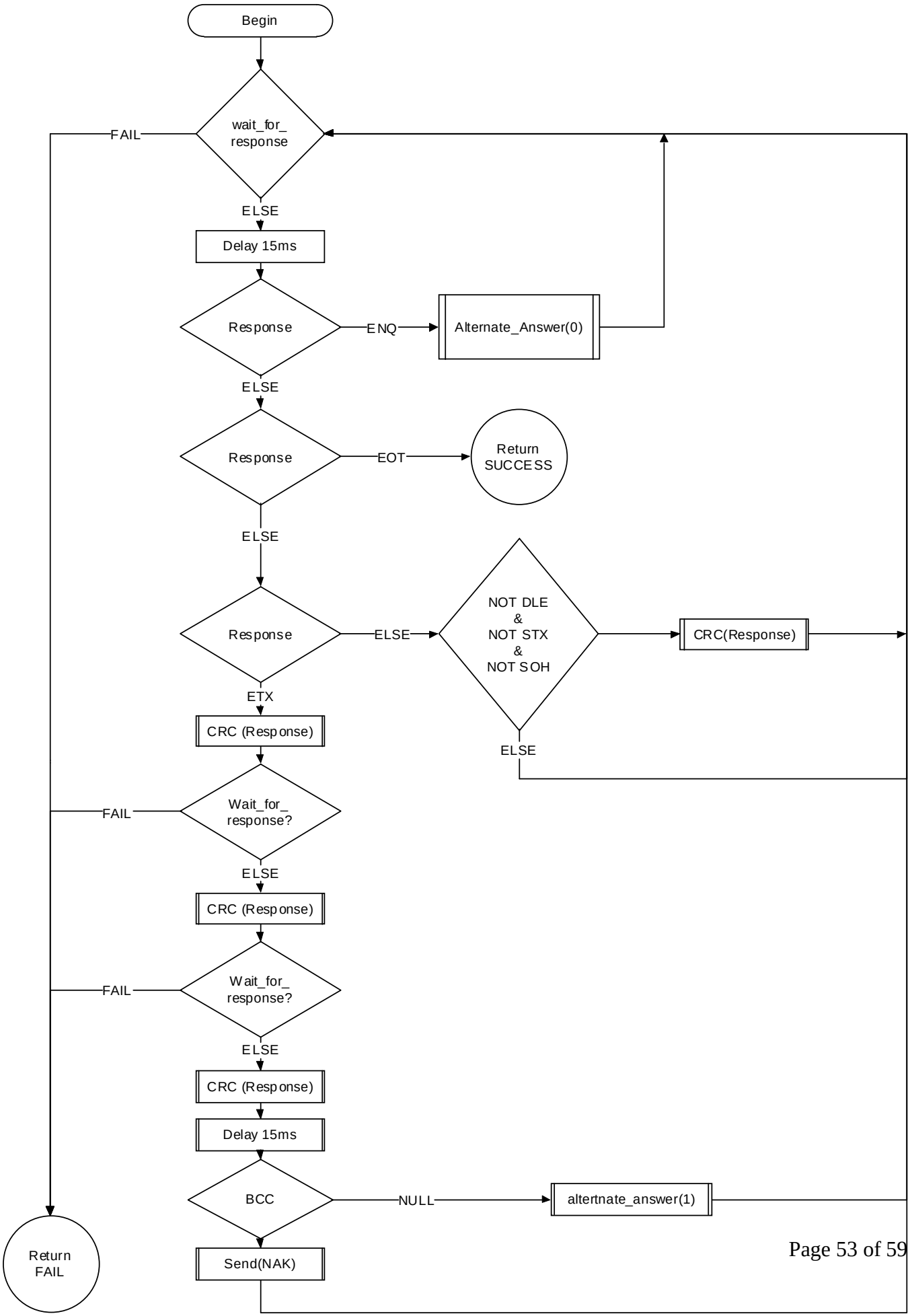



Begin
FlushCom()
Send(ENQ)
ch = Wait(200)
if ch = ENQ
TRUE
dex_com = MASTER
Slave Handshake
SUCCESS
Master Handshake
SUCCESS
Third Handshake
Return SUCCESS
FALSE
FALSE
if ch = DLE
TRUE
Wait for response
SUCCESS
dex_com = SLAVE
Master Handshake
SUCCESS
Slave Handshake
SUCCESS
Third Handshake
Return SUCCESS
FAIL
dex_com = NAK
DEX Communication Failed

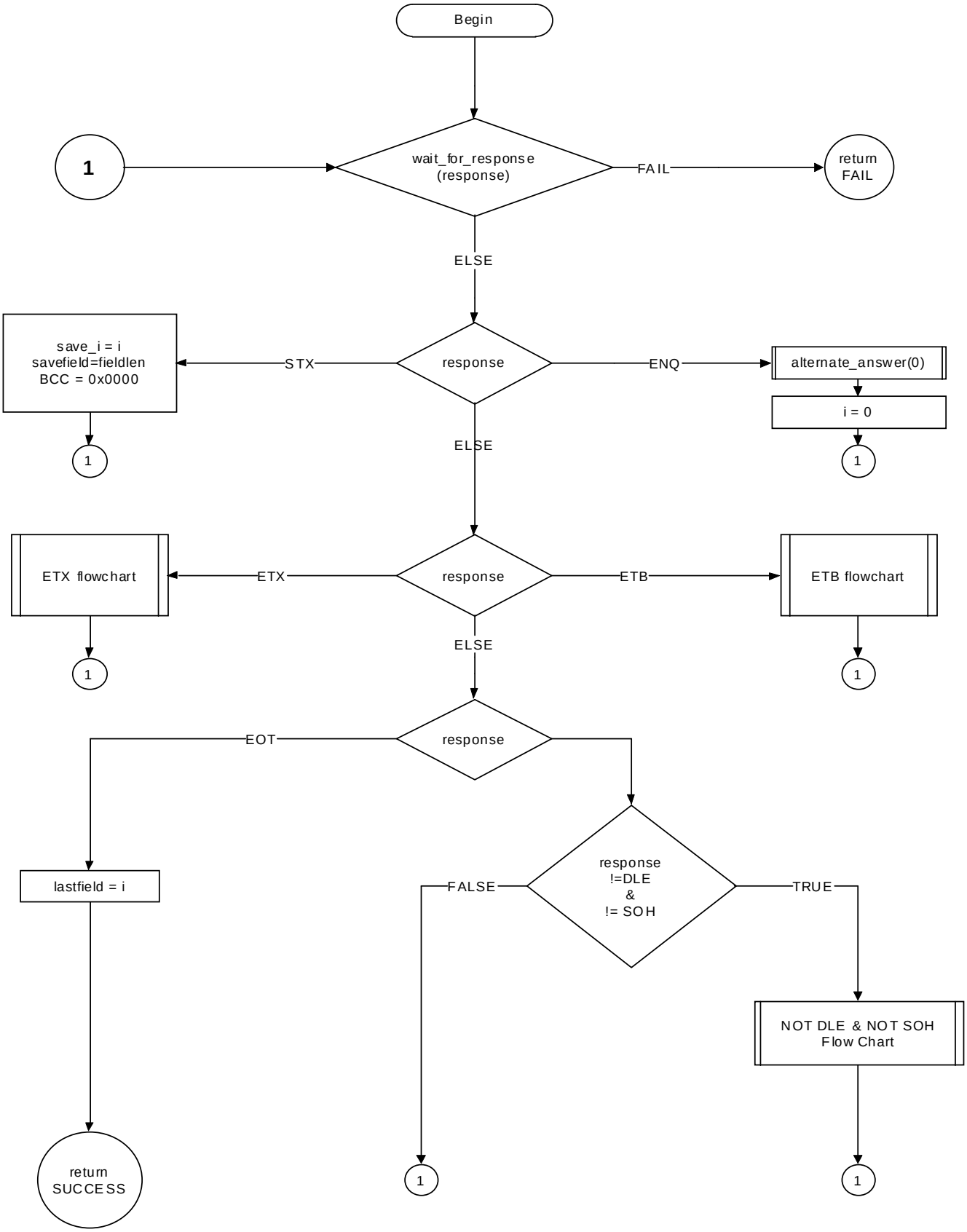
Master Handshake



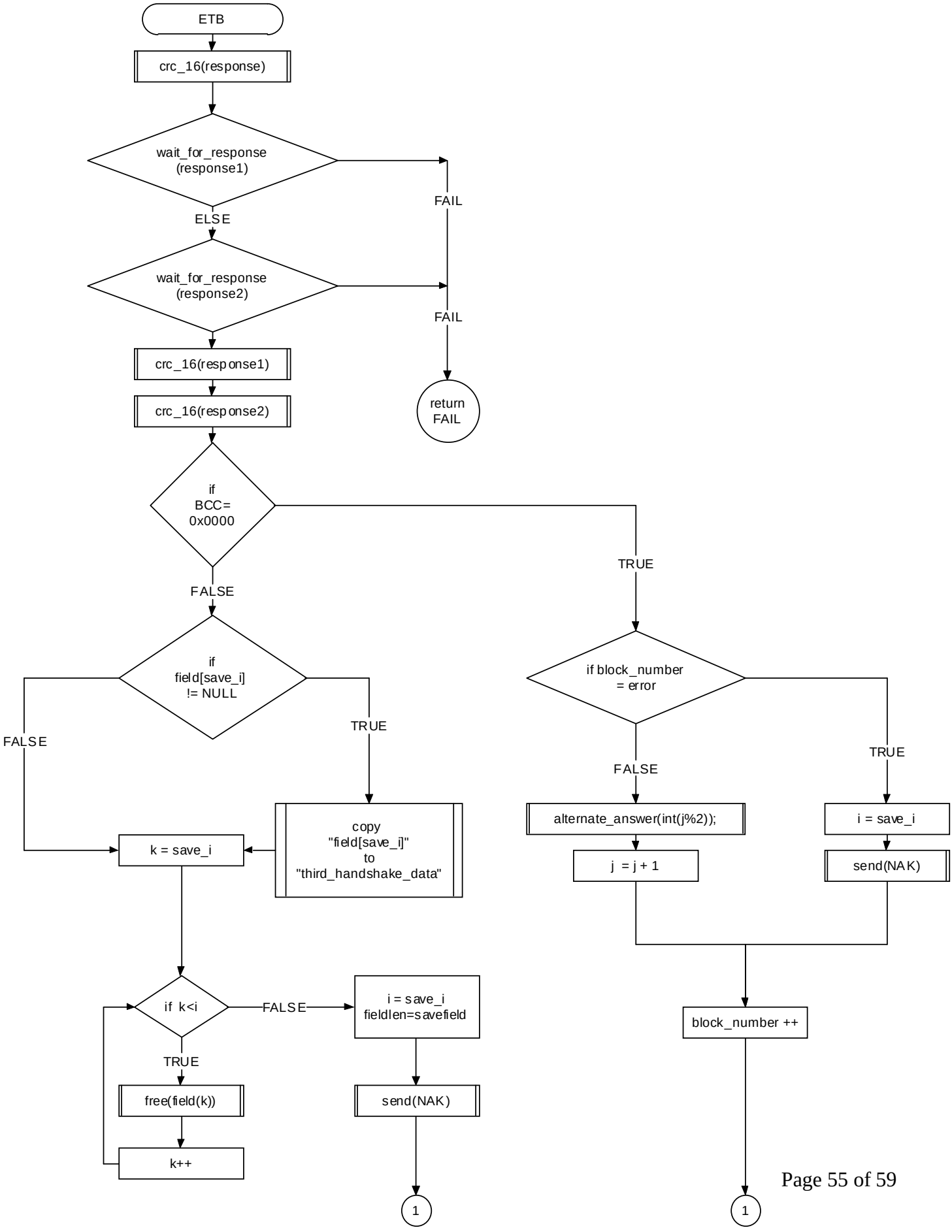
Slave Handshake



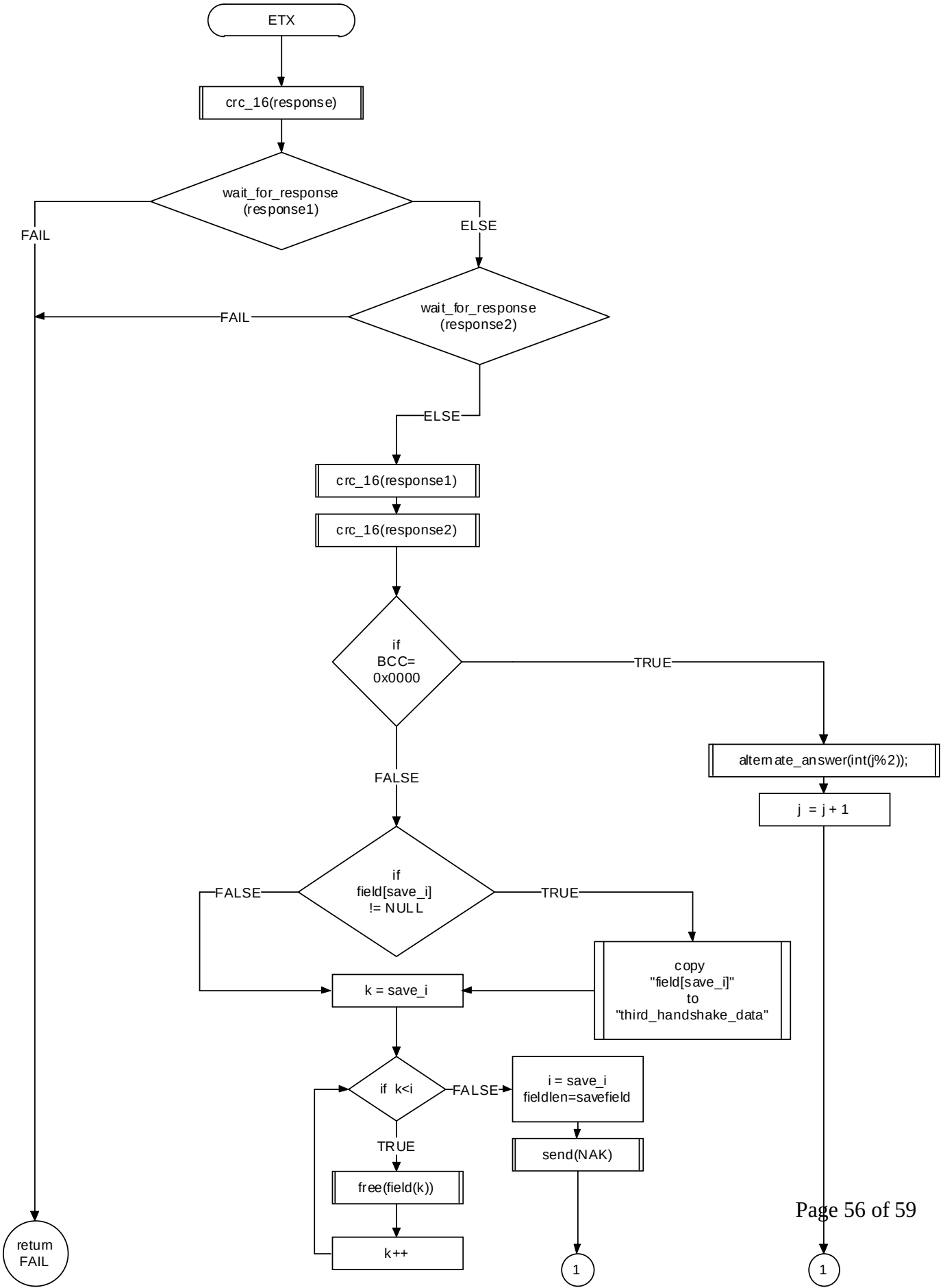
Third Handshake



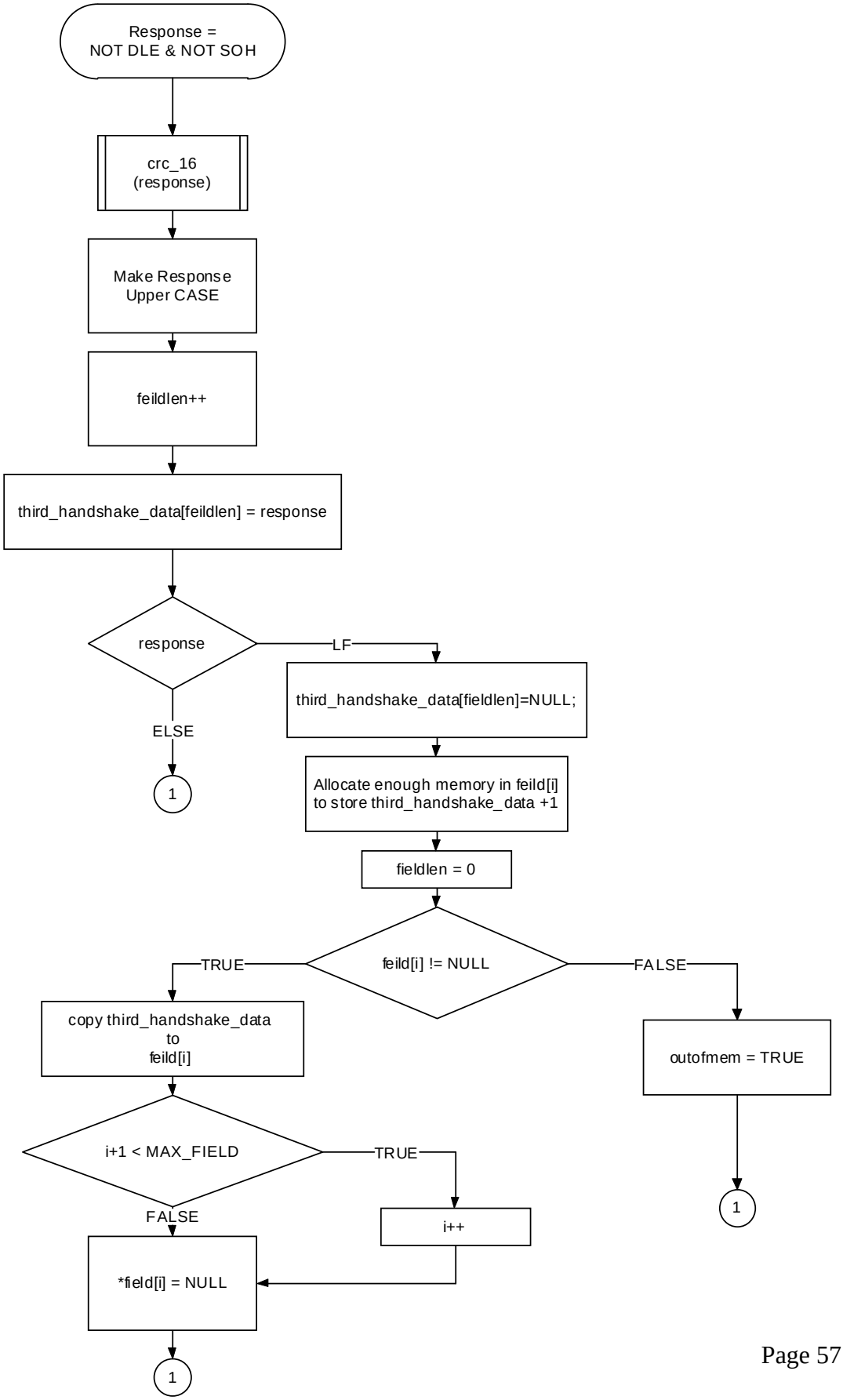
Third Handshake - ETB



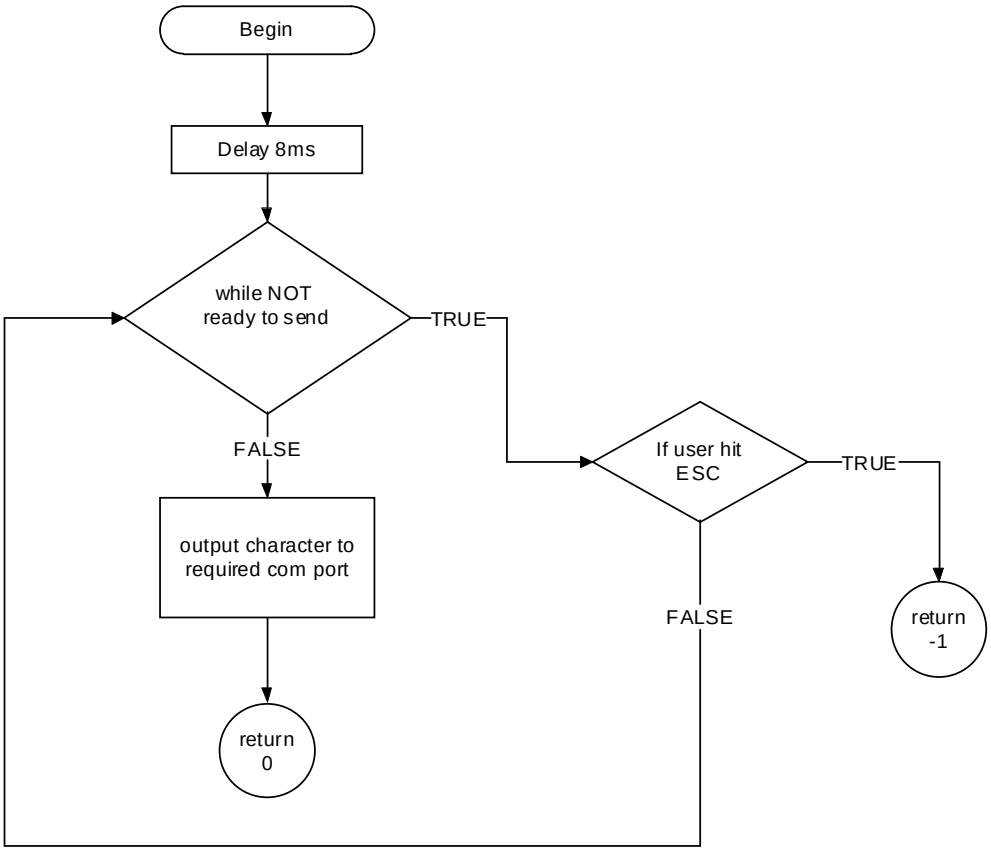
Third Handshake - ETX



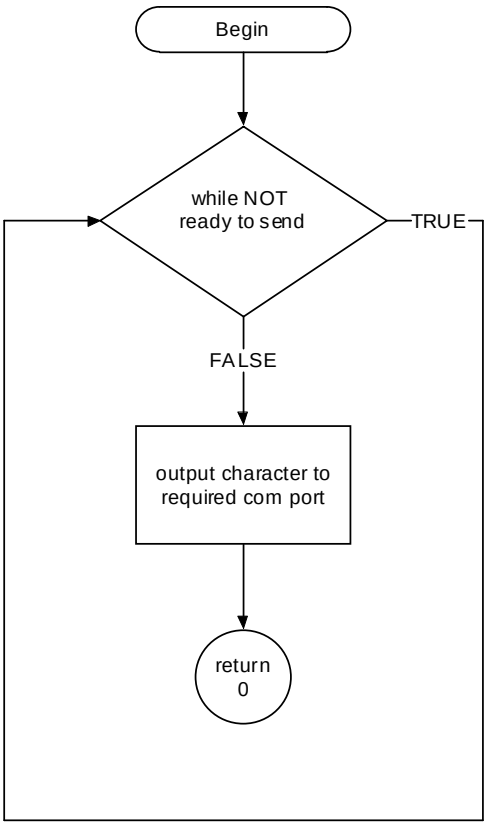
Third Handshake - NOT DLE & NOT SOH



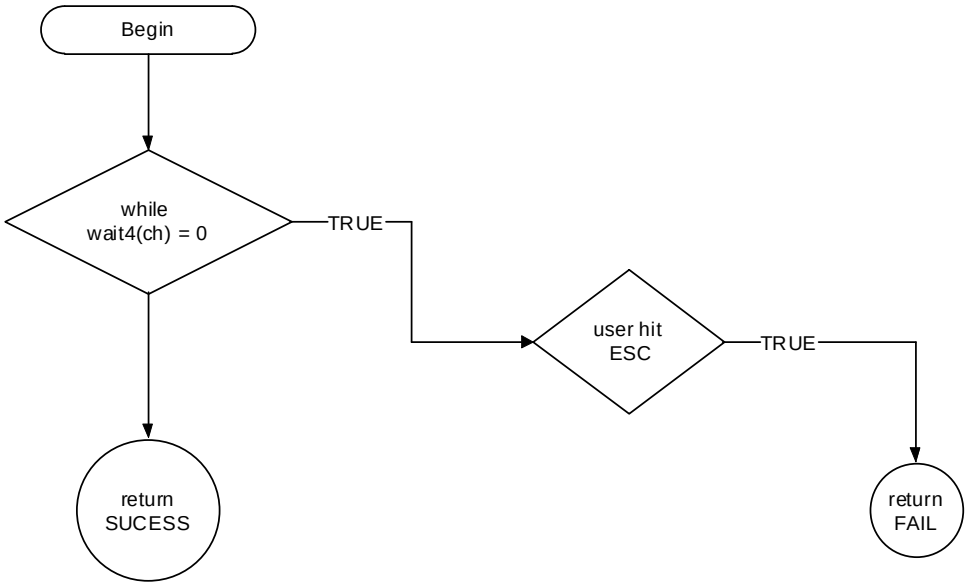
send



send_no_delay



wait_for_response



wait4

Functions for reading DEX audit data